VRaptor

Desenvolvimento Ágil para Web com Java

LUCAS CAVALCANTI

Sumário

Capítulo 1

Introdução

1.1 Um pouco de história

Há muito tempo, desenvolver uma aplicação web em Java era uma tarefa muito trabalhosa. Por um lado, tínhamos os Servlets e JSP, nos quais todo o tratamento das requisições era manual e cada projeto ou empresa acabava criando seu próprio *framework* — "O" framework — para trabalhar de uma forma mais agradável. Por outro lado, tínhamos vários frameworks que se propunham a resolver os problemas da dificuldade de se trabalhar com os Servlets, mas que lhe obrigavam a escrever toneladas de XML, arquivos `.properties` e a estender classes ou implementar interfaces para poder agradar a esse framework.

Nesse cenário surgiu a ideia de um framework mais simples, que facilitasse o desenvolvimento web sem tornar o projeto refém das suas classes e interfaces. Surgia o VRaptor, criado pelos irmãos Paulo Silveira e Guilherme Silveira, em 2004 na Universidade de São Paulo. Em 2006 foi lançada a primeira versão estável: o VRaptor 2, com a ajuda do Fabio Kung, do Nico Steppat e vários outros desenvolvedores, absorvendo várias ideias e boas práticas que vieram do Ruby on Rails.

Em 2009, o VRaptor foi totalmente reformulado, levando em conta a experiência obtida com os erros e acertos da versão anterior e de muitos outros frameworks da época. A versão 3 leva ao extremo os conceitos de Convenção sobre Configuração e Injeção de Dependências, em que todos os comportamentos normais podem ser mudados facilmente usando anotações ou sobrescrevendo um dos componentes internos — bastando, para tanto, implementar uma classe com a interface desse componente. Além disso, esta nova versão possibilita criar de uma maneira fácil não apenas aplicações web que rodam no *browser*, mas também serviços web que seguem as ideias de serviços RESTful, facilitando a comunicação entre sistemas e a implementação de AJAX no browser.

1.2 Onde você vai chegar com esse livro?

O objetivo desse livro é mostrar muito além do uso básico do VRaptor e suas convenções. Mais importante que aprender essas convenções é entender a arquitetura interna do VRaptor e como criar suas próprias convenções, adaptando o VRaptor para as necessidades específicas do seu projeto. Saber como as coisas funcionam internamente é metade do caminho andado para que você tenha um domínio sobre o framework e se sinta confortável para fazer o que quiser com ele.

Durante esses mais de três anos do VRaptor 3, tenho respondido as mais variadas dúvidas sobre o seu uso no GUJ (http://www.guj.com.br) , o que nos ajudou muito a modelar as novas funcionalidades e descobrir *bugs*. Mas, muito mais do que isso, mostrou-nos todo o poder da extensibilidade: mesmo os problemas mais complexos e necessidades mais específicas dos projetos conseguiram ser resolvidos sobrescrevendo o comportamento do VRaptor usando os meios normais da sua API ou sobrescrevendo um de seus componentes.

Neste livro você vai ver o uso básico e esperado, junto com o que considero boas práticas e uso produtivo das ferramentas que esse framework proporciona. Mas também compartilhar vários casos do uso não esperado e como também é fácil implementá-los.

Portanto, esse livro é pra quem quer entender como funciona o VRaptor e como aproveitar todo o seu poder e sua extensibilidade para tornar o desenvolvimento de aplicações o mais fácil e produtivo quanto possível.

Está pronto para começar?

Capítulo 2

O começo de um projeto com VRaptor

Um grande problema no desenvolvimento de um projeto é justamente como (e quando) começá-lo. Isso não só na parte técnica, é preciso definir muito bem qual é o problema a ser resolvido, qual é o perfil do usuário do sistema, como será a usabilidade, onde será instalado, e muitos outros detalhes que devem influenciar nas escolhas que você deve fazer.

Hoje em dia temos muitas ferramentas — bibliotecas, frameworks, linguagens de programação, servidores etc — à nossa disposição, o que torna muito difícil definir o que usar. Muitas destas ferramentas são boas e adequadas para resolver o seu problema. Mais importante do que saber quando usar uma ferramenta é saber quando não usá-la, ou quando ela vai atrapalhar mais do que ajudar.

2.1 Vantagens e desvantagens

O VRaptor é um framework feito para desenvolver aplicações Web e também muito bom para desenvolver APIs HTTP/REST para comunicação entre sistemas. É um framework MVC que dá um suporte muito bom à parte do Modelo — as classes que contém a lógica de negócio da sua aplicação. Com ele, é muito fácil aplicar as melhores práticas de orientação a objetos, já que não impõe restrições no *design* das suas classes, além de facilitar a prática de Inversão de Controle por Injeção de Dependências.

É também um dos frameworks mais extensíveis que existe em Java. Praticamente todos os comportamentos do VRaptor podem ser sobrescritos com muito pouco código e sem configurações em XML ou arquivos `.properties`.

Além disso, é um framework que nos deixa totalmente livre para escolher sua camada de Visualização. O que também significa que não nos ajudará muito para desenvolver o HTML das páginas, diferente de alguns outros frameworks como, por exemplo, o JSF. Você pode usar JSP, Velocity, Freemarker para gerar os templates, mas os componentes visuais precisam ser feitos manualmente, ou usando alguma biblioteca externa como o jQuery UI, YUI ou ExtJS. Cada vez mais, essas bibliotecas se tornam mais simples de serem usadas, então até mesmo com pouco conhecimento de Javascript, HTML e CSS, é possível criar uma interface rica e interessante.

2.2 O projeto — Loja de livros

Nas próximas páginas, desenvolveremos uma loja de livros online que terá duas partes: a administrativa, na qual serão gerenciados todos os livros, quantidade em estoque etc; e a parte de vendas, em que serão mostrados todos os livros para que o usuário possa escolhê-los e comprá-los.

Essas duas partes serão feitas em projetos separados, que trocarão dados sempre que necessário. A parte administrativa guardará todos os dados referentes aos livros, num banco de dados. A parte de vendas não terá acesso a esse banco de dados, apenas consultará os dados do livro usando a API da parte administrativa.

A ideia desse projeto é ser simples o suficiente para que consigamos começar a desenvolvê-lo sem muito contexto inicial, mas ao mesmo tempo tratando de diversos aspectos reais do desenvolvimento de aplicações web, como cadastros, uploads de imagens, integração entre sistemas, comunicação com o banco de dados, transações, segurança, entre outros.

2.3 Criando os projetos

Existem várias formas de criar um projeto com VRaptor, cada uma com seus pré-requisitos e flexibilidades. Portanto, mostraremos a seguir as maneiras mais comuns e suas limitações.

VRaptor Blank Project

O VRaptor Blank Project é um projeto preparado com o mínimo necessário para rodar o VRaptor. Para baixá-lo, iremos na página de *Downloads* do site VRaptor: http://vraptor.caelum.com.br link *Download*. Esse projeto está em um zip com o nome `vraptor-blank-project-3.x.y.zip`, no qual 3.x.y é a última versão do VRaptor e já está configurado para Eclipse e Netbeans, bastando importar o zip para poder rodá-lo. No caso do Eclipse:

1) Acesse o menu **File** >> **Import**;
2) Selecione **Existing projects into Workspace**;
3) Marque a opção **Select archive file** e clique em **Browse...**;
4) Escolha o **vraptor-blank-project-3.x.y.zip**;
5) Clique em **Finish**. Um projeto com o nome **vraptor-blank-project** deve aparecer;
6) Clique com o botão direito no projeto e escolha **Run As** >> **Run On Server**;
7) Escolha algum servidor existente ou crie um;
8) Deve aparecer uma tela dizendo: "It works!! VRaptor! /vraptor-blank-project/";
9) O projeto com o VRaptor está configurado com sucesso!

A partir de agora, você pode desenvolver o seu projeto. Essa é a maneira mais rápida de ver um projeto funcionando com o VRaptor, mas dá um trabalho um pouco maior para transformar em um outro projeto. Isso, no Eclipse, envolve renomear o `vraptor-blank-project` para `seu-projeto` e ajustar algumas configurações.

Zip do VRaptor

Também na página de Downloads do VRaptor existe um arquivo chamado `vraptor-3.x.y.zip`, que contém a distribuição completa da última versão do VRaptor. Nesse zip podemos encontrar o jar do vraptor, suas dependências, sua documentação, javadoc e código fonte. Assim, já é possível linkar esses artefatos na sua IDE [footnote Eclipse, Netbeans etc] e facilitar o desenvolvimento.

Para criar um projeto usando esse zip, você precisa criar um novo projeto web na sua IDE. No caso do Eclipse:

1) Crie um novo projeto web. Aperte **Ctrl+N** e escolha **Dynamic Web Project**;
2) Escolha um nome de projeto, por exemplo **livraria**;
3) Selecione como **Target Runtime** um servidor compatível com Servlet 3.0, como o Tomcat 7;
4) Selecione **3.0** como **Dynamic web module version**;
5) Clique em **Finish**;
6) Abra o zip do VRaptor e copie o arquivo vraptor-3.x.y.jar para a pasta `WebContent/WEB-INF/lib`;
7) Também no zip do VRaptor, abra a pasta **lib/** e copie todos os jars para `WebContent/WEB-INF/lib`;
8) Atualize o projeto (`F5`) e rode esse projeto no servidor;
9) Faça o `Hello World`, seguindo os passos da seção 2.4;
10) Projeto configurado com sucesso!

Rodando o VRaptor em ambientes Servlet 2.5

As configurações acima são para ambientes `Servlet 3.0`. Se quiser rodar o VRaptor em um servidor `Servlet 2.5`, como o Tomcat 6, precisa também acrescentar no `web.xml` as seguintes linhas:

```
<filter>
    <filter-name>vraptor</filter-name>
    <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
</filter>

<filter-mapping>
    <filter-name>vraptor</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Esse processo é um pouco mais manual, mas é bem mais flexível e não precisa de nada instalado a mais no sistema.

Maven

Outro bom jeito de começar um projeto java é usando o Maven, uma ferramenta de build e gerenciamento de dependências. Os jars do VRaptor estão no repositório do Maven, portanto podemos acrescentá-lo como dependência no Maven. Para isso, precisamos dele instalado no computador (http://maven.apache.org/download.html) , ou na IDE (http://maven.apache.org/eclipse-plugin.html ou http://maven.apache.org/netbeans-module.html) . Para criar um projeto VRaptor com o Maven siga os passos:

1) Rode o plugin de archetype do Maven. Na linha de comando:

   ```
   mvn archetype:generate
   ```

   No Eclipse: **Ctrl+N**, escolha **Maven Project** e aperte **Next** até a tela de seleção do archetype.

2) Selecione o **maven-archetype-webapp** (na linha de comando você pode digitar isso em vez do número)

3) Escolha um `groupId` (ex. `br.com.casadocodigo`), um `artifactId` (ex. `livraria`) e um pacote (ex. `br.com.casadocodigo.livraria`).

4) No **pom.xml** do projeto criado, adicione a dependência do VRaptor (editando o xml ou pelos wizards do eclipse):

```
<project ....>
  <dependencies>
    ...
    <dependency>
        <groupId>br.com.caelum</groupId>
        <artifactId>vraptor</artifactId>
        <version>3.5.1</version><!-- ou a última versão -->
    </dependency>
  </dependencies>
</project>
```

5) Faça o **Hello World**, na seção 2.4.

6) Projeto criado com sucesso!

O Maven é muito produtivo se você quiser trabalhar do jeito dele, mas se precisar fazer algo mais personalizado para seu projeto, prepare-se para se divertir a valer com toneladas de configurações em XML. E também reserve um espaço no seu HD pra fazer o backup da internet a cada comando.

VRaptor Scaffold

O **VRaptor Scaffold** é uma extensão criada pelo Rodolfo Liviero (http://rodolfoliviero.com.br) , inspirado pela funcionalidade de *scaffolding* do **Ruby on Rails**. Com ele, a criação de projetos com o VRaptor fica bem mais simples e flexível. A documentação completa está em: http://vraptor.caelum.com.br/pt/docs/vraptor-scaffold-pt/. Para usá-lo, é necessário ter o RubyGems(http://rubygems.org) instalado no sistema e seguir os seguintes passos:

1) Instale a gem vraptor-scaffold:

```
gem install vraptor-scaffold
```

2) Crie um novo projeto:

```
vraptor new livraria --package=br.com.casadocodigo.livraria
```

3) Entre na pasta do projeto e baixe as dependências com:

```
cd livraria
ant resolve
```

4) Importe o projeto no Eclipse (ou Netbeans)

5) Faça o HelloWorld (seção 2.4) e pronto!

O projeto criado usa o ant (http://ant.apache.org) para gerenciar o build e o ivy (http://ant.apache.org/ivy) para gerenciar as dependências. Mas é possível mudá-lo para usar o Maven ou o Gradle como ferramenta de build.

O vraptor-scaffold gera vários arquivos no projeto, entre eles:

- arquivos de configuração de projeto do eclipse (.project, .classpath e .settings)
- **build.xml** e **build.properties** — arquivos de configuração do ant com algumas targets já configuradas, como **compile** para compilar as classes, **war** para gerar o war e **jetty.run** para subir a aplicação num jetty.
- **ivy.xml** e **ivy.jar** — arquivos de configuração do ivy. Edite o arquivo ivy.xml para acrescentar dependências ao projeto e rode o `ant resolve` para baixá-las.
- em **src/main/java** foram criados três pacotes: `controllers`, `models` e `repositories`, e arquivos padrão se você usar o comando **vraptor scaffold** para criar os modelos da sua aplicação.
- em **src/main/resources** foram criados os arquivos de configuração do hibernate, jpa e log4j.
- em **src/main/webapp** foram criados arquivos javascript e css padronizados para facilitar o início do projeto. Também está configurado o sitemesh (http://sitemesh.org) , que facilita a criação de templates (como cabeçalhos e rodapés, menus padronizados etc).

2.4 Criando um Hello World

Com o projeto criado, podemos ver o VRaptor funcionando rapidamente, apenas usando as suas convenções básicas. O objetivo do VRaptor é se intrometer o mínimo possível nas classes da sua aplicação, então vamos criar uma classe java normal, que faça um simples "Olá mundo":

```
public class Mundo {

    public void ola() {
        System.out.println("Olá Mundo!")
    }
}
```

Quando estamos em uma aplicação web, queremos poder executar algum código quando o usuário faz uma requisição ao sistema — ao acessar uma URL no navegador, por exemplo. Para que o método `ola()` seja acessível por uma URL, tudo que precisamos fazer é anotar a classe `Mundo` com `@Resource`:

```
@Resource
public class Mundo {
    public void ola() {
        System.out.println("Olá Mundo!")
    }
}
```

Seguindo a convenção do VRaptor, podemos acessar esse método pela URL http://localhost:8080/livraria/mundo/ola. As classes que recebem requisições web no nosso sistema têm um papel especial na arquitetura **MVC** — são os **Controllers**. Para deixarmos mais claro esse papel da nossa classe `Mundo`, podemos dizer isso no nome da classe, ou seja, `MundoController`.

O VRaptor exclui o sufixo `Controller` das convenções para definir a URL, podemos renomear nossa classe e tudo continuará funcionando:

```
@Resource
public class MundoController {
    public void ola() {
        System.out.println("Olá Mundo!")
    }
}
```

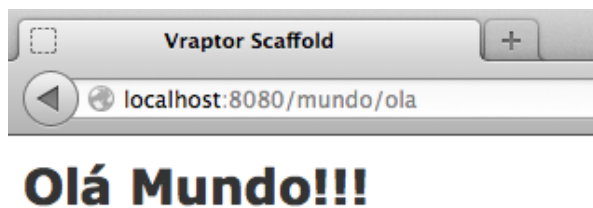
Dessa forma, continuamos acessando o método `ola()` pela URL http://localhost:8080/livraria/mundo/ola. Esse método imprime "Olá Mundo!", só que no console do servidor, precisamos mostrar uma tela para o usuário que acessou a URL do método e, seguindo o padrão MVC, geramos essa tela em outro arquivo, por exemplo um JSP. Essa é a parte **V** do MVC, a visualização. Por padrão, ao final da invocação do método `ola`, o VRaptor redireciona a requisição para o jsp em `/WEB-INF/jsp/mundo/ola.jsp`.

Os jsps ficam em `/WEB-INF/jsp` para não serem acessíveis diretamente pelo navegador, forçando todas as requisições a passarem pelo ciclo completo do *MVC*. Depois disso, vem o nome do controller (sem a palavra Controller) seguido pelo nome do método e a extensão jsp. Todas essas convenções podem ser configuradas e modificadas e veremos mais disso adiante.

Criando então a jsp correspondente ao método `ola()` do `MundoController`, em `/WEB-INF/jsp/mundo/ola.jsp`:

```
<html>
  <h1>Olá Mundo!!!</h1>
</html>
```

Para ver tudo funcionando, vamos subir o servidor e acessar http://localhost:8080/livraria/mundo/ola. A página que acabamos de criar deverá aparecer:

Olá Mundo!!!

E pronto! Temos todo o código necessário para processar uma requisição com o VRaptor. Agora vamos criar os projetos necessários para criar nossa livraria.

2.5 Criando os projetos da livraria

O nosso sistema será dividido em duas partes, uma que cuidará do site onde os usuários comprarão os livros — o projeto `livraria-site` — e outra que cuidará do

cadastro dos livros — o projeto `livraria-admin`. Além disso, usaremos o Tomcat como servidor e o MySQL como banco de dados.

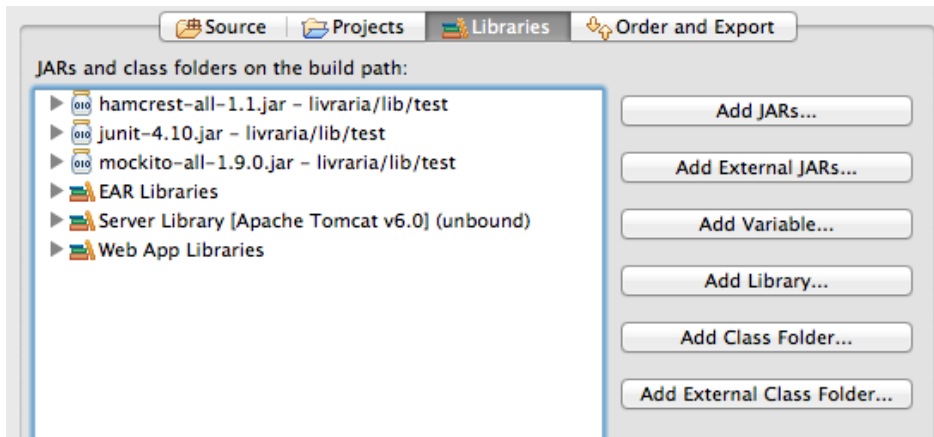
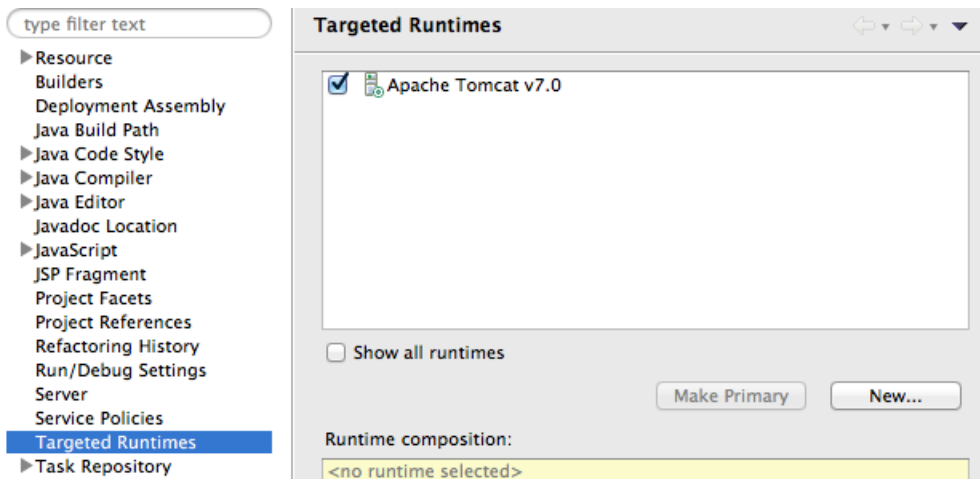
Para criar esses projetos, podemos usar qualquer um dos métodos descritos na seção 2.3, mas para facilitar um pouco a configuração inicial vamos usar o `vraptor-scaffold`. Na sua pasta `workspace`, execute:

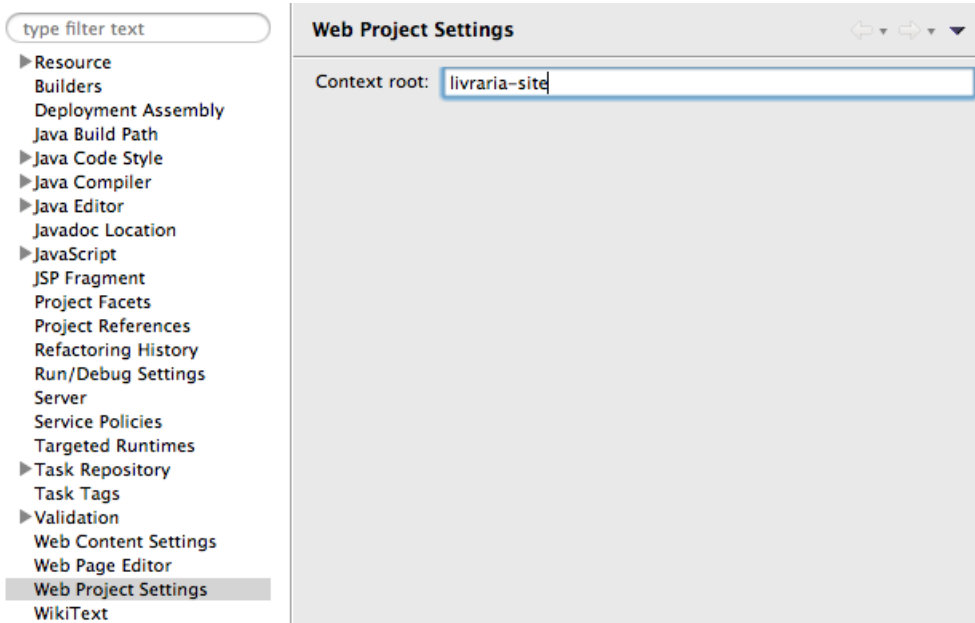
```
vraptor new livraria-site
vraptor new livraria-admin

cd livraria-site
ant resolve

cd ../livraria-admin
ant resolve
```

Caso não possa ou não queira instalar o `vraptor-scaffold` na sua máquina, você pode baixar os projetos gerados em: https://github.com/lucascs/livraria/archive/projetos-criados.zip

Você também pode acompanhar a evolução do projeto navegando pelos commits desse repositório do GIT: https://github.com/lucascs/livraria/commits/master

Agora, podemos importar esses dois projetos no Eclipse, usando o menu `File >> Import >> Existing projects into workspace`. Selecione a sua pasta `workspace`, onde você rodou os comandos do *scaffold*, e selecione os projetos em seguida.

Provavelmente será necessário corrigir o `build path` dos projetos para apontar para o java e o Tomcat configurados no seu computador. No caso do Eclipse, clique com o botão direito em cada um dos projetos e vá em **Build Path** >> **Configure Build Path** e selecione a aba **Libraries**.

Clique em **Add Library**, selecione **JRE System Library** e selecione o padrão do seu sistema. Se a **Server Library** estiver *(unbound)* como na imagem, clique em cima dela, depois em **Edit** e selecione um dos servidores instalados no seu eclipse. Se não houver nenhum, é possível criar um no menu **New** >> **Server** (Ctrl + N e escreva server), seguindo os passos do wizard que apareceu. Também será necessário alterar o **Targeted Runtime** dos projetos para o servidor escolhido acima, ou seja, o servidor onde os projetos vão rodar.

E para que seja possível colocar os dois projetos no mesmo servidor, precisamos configurar o contexto raiz de cada projeto. Ainda nas propriedades de cada projeto e

então em **Web Project Settings** vamos mudar o **Context root:** para o nome de cada projeto, ou seja, **livraria-admin** e **livraria-site**.

Nesse sistema vamos usar a JPA, a especificação de persistência do java, para guardar os dados da nossa aplicação no banco de dados, com o Hibernate servindo como implementação da JPA. O VRaptor Scaffold já gera um esqueleto da configuração necessária para que o Hibernate funcione com a JPA, juntamente com as dependências declaradas no ivy.xml que já foram baixadas quando rodamos o comando `ant resolve`.

As configurações são o arquivo `src/main/resources/META-INF/persistence.xml`, no qual declaramos o `persistence-unit default` para que possamos trabalhar com a JPA, e o arquivo `src/main/resources/hibernate.properties`, no qual colocamos as credenciais do banco de dados e demais configurações do hibernate. Só iremos usar banco de dados no projeto **livraria-admin**, portanto podemos apagar esses arquivos de configuração do projeto **livraria-site**.

O **hibernate.properties** gerado pelo **vraptor-scaffold** vem configurado para usar o banco de dados **HSQLDB**, que é um banco bem simples, bastante útil para fazermos testes na nossa máquina de desenvolvimento, mas não deve ser usado em sistemas de produção. Esse banco é o suficiente para continuarmos o desenvolvi-

mento dos projetos deste livro, pois usaremos apenas consultas simples ao banco de dados, mas se quisermos usar outro banco de dados, como o MySQL, o Postgres ou o Oracle, é necessário seguir os passos abaixo:

- Mude as propriedades de conexão com o banco, as que começam com `hibernate.connection`, para as do banco escolhido. No caso do MySQL são:

  ```
  hibernate.connection.driver_class=com.mysql.jdbc.Driver
  hibernate.connection.url=jdbc:mysql://localhost/livraria
  ```

  O **localhost** e **livraria** são, respectivamente, o endereço da máquina do banco e o nome da base de dados. Essa base de dados precisa ser criada antes de ser usada no projeto.

- Configure as credenciais do usuário. No MySQL o usuário padrão é **root** e a senha vazia:

  ```
  hibernate.connection.username=root
  hibernate.connection.password=
  ```

  Use o usuário e a senha que você configurou para a base de dados criada.

- Adicione o jar do driver do banco de dados. No nosso caso, basta adicionar no no ivy.xml do projeto **livraria-admin**:

  ```
  <dependency org="mysql" name="mysql-connector-java"
        rev="5.1.15" conf="default"/>
  ```

  E rodar o comando `ant resolve` nesse projeto.

Para finalizar a configuração dos projetos, vamos apagar os arquivos do pacote **app** das pastas **src/main/java** e **src/test/java** dos dois projetos, pois vamos criar todos os outros arquivos da aplicação sem a ajuda do **vraptor-scaffold**.

Agora sim, podemos continuar o desenvolvimento das duas partes do nosso sistema: o **livraria-admin** e o **livraria-site**, que veremos nos próximos capítulos, a começar pelo cadastro de livros, no admin.

Capítulo 3

Crie o seu primeiro cadastro

Agora que já temos o projeto criado, completamente configurado e já fizemos até um "olá mundo" para garantir que tudo funciona, podemos começar a implementar de fato nossa aplicação. Vamos fazer a criação do cadastro principal, com o qual iremos poder gravar e manipular as informações dos livros.

Passaremos por diversas características do framework e veremos todas elas em detalhes no decorrer do livro. O mais importante é que agora você perceba a facilidade e objetividade do framework. Preparado?

3.1 Criação dos modelos

O nosso sistema será uma loja de livros, logo um dos dados mais importantes que teremos que guardar é a coleção de livros do sistema. Cada livro terá um título, uma descrição, um ISBN, um preço e uma data de publicação. Dando uma atenção especial aos dois últimos atributos:

- `preço`: um valor em dinheiro. Vamos representá-lo com o tipo `BigDecimal`, com o qual podemos manter precisão que a gente escolher pra ele e que também possui algumas facilidades se precisarmos realizar cálculos.
- `data de publicação`: Podemos representar essa data usando as classes já embutidas no java: `Date` ou `Calendar`. Escolheremos nesse momento a classe `Calendar`, por ser um pouco mais completa, possuindo métodos para manipular a data sem ter que ficar fazendo contas com timestamps.

Melhor alternativa para as datas

Tanto a classe `java.util.Date` como `java.util.Calendar` possuem dois problemas fundamentais: a interface dos objetos é muito ruim de se trabalhar e elas só conseguem representar data e hora. Para resolver esse problema, existe um projeto chamado Joda-time (http://joda-time.org) , que possui representações bem mais completas e flexíveis para abstrações de tempo. Por exemplo, no nosso caso só precisamos da data, sem a hora, então poderíamos usar a classe `LocalDate` do Joda-time.

Vamos criar a classe que representa esse livro no projeto **livraria-admin**.

```
package br.com.casadocodigo.livraria.modelo;

import java.math.BigDecimal;
import java.util.Calendar;

public class Livro {
    private String titulo;
    private String descricao;
    private String isbn;
    private BigDecimal preco;
    private Calendar dataPublicacao;

    //getters/setters
}
```

Para cadastrar os livros no nosso sistema, precisamos armazená-los em algum lugar. No momento, a forma com a qual vamos conseguir fazer esse armazenamento

não importa muito, portanto vamos criar uma classe que representa um conjunto de livros — um *repositório*, conforme explicado no livro Domain Driven Design do Eric Evans. Na vida real, guardamos livros numa estante, então nada mais justo que usar esse nome para abstrair o lugar onde guardamos livros. Criaremos uma interface com todas as operações que queremos ter, já que ainda não sabemos ainda como será a implementação.

```
package br.com.casadocodigo.livraria.modelo;

public interface Estante {

   void guarda(Livro livro);

   List<Livro> todosOsLivros();
}
```

3.2 Criando o Cadastro

Para criar um livro agora no nosso projeto web, precisamos criar uma página com o formulário de adição do livro.

```
<form action="?" method="post">
    ...
</form>
```

Formulários como esse podem conter alguma parte mais dinâmica, por exemplo a lista de categorias em que um livro pode estar. Por esse motivo, sempre que criamos uma página, passamos primeiro por uma classe java que vai popular os dados necessários para ela. Criaremos então um Controlador, que vai controlar o nosso cadastro de livros, esse será o nosso `LivrosController`, com um método para fornecer acesso ao formulário.

```
package br.com.casadocodigo.livraria.controlador;

public class LivrosController {

    public void formulario() {}

}
```

Para que o VRaptor passe a gerenciar essa classe como um controlador, precisamos anotá-la com `@Resource`. Isso significa que essa classe controla um dos recursos da aplicação, no caso os livros. Chamamos de recursos os dados que a aplicação produz e/ou gerencia e, ao colocarmos o `@Resouce` em uma classe dizemos ao VRaptor que essa classe será o ponto de acesso via Web de algum dos recursos do sistema.

```
import br.com.caelum.vraptor.Resource;

@Resource
public class LivrosController {

    public void formulario() {}

}
```

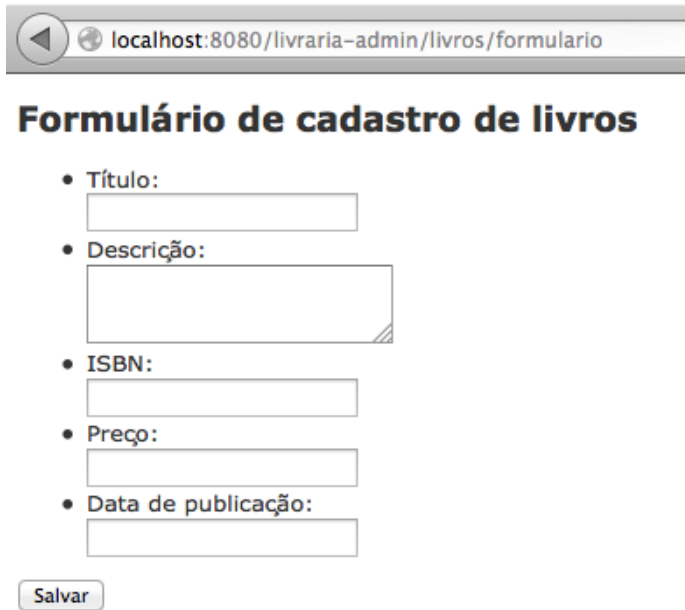
A partir desse momento, seguindo as convenções do VRaptor, conseguimos executar o que está dentro do método `formulario`, acessando a URI `/livros/formulario`. Lembrando que a convenção é: `/<nome_do_recurso>/<nome_do_metodo>`. O nome do recurso é o nome da classe sem a palavra `Controller`, **Livros**, com a primeira letra minúscula: `livros`. O nome do método é exatamente o nome do método. O caminho completo no nosso caso seria http://localhost:8080/livraria-admin/livros/formulario.

O método `formulario` não faz nada de interessante ainda, mas vamos usá-lo para chegar até a página do formulário que, segundo a convenção do VRaptor, estará em `WEB-INF/jsp/livros/formulario.jsp`. Essa convenção é parecida com a da URI: `WEB-INF/jsp/<nome_do_recurso>/<nome_do_metodo>.jsp`. O caminho completo para a jsp seria:

```
livraria-admin/src/main/webapp/WEB-INF/jsp/livros/formulario.jsp
```

Se o projeto foi criado direto pelo Eclipse, sem o maven ou o vraptor-scaffold, em vez de `src/main/webapp` use `WebContent`.

Podemos seguir com a criação do formulário. Precisamos criar um objeto livro a partir dele, e para isso conseguimos usar mais uma convenção do VRaptor no nome dos inputs. Para isso, precisamos de um nome de variável para o livro criado, por exemplo, `livro`. Cada input deve ter seu atributo `name` começando com o nome dessa variável, e cada propriedade acessível delimitada por pontos. Para popular o

título do livro, usamos `livro.titulo`, para o preço, `livro.preco` e assim por diante:

```
<form method="post">
  <h2>Formulário de cadastro de livros</h2>
  <ul>
    <li>Titulo: <br/>
        <input type="text" name="livro.titulo" /></li>

    <li>Descricao: <br/>
        <textarea name="livro.descricao"></textarea></li>

    <li>ISBN: <br/>
        <input type="text" name="livro.isbn" /></li>

    <li>Preco: <br/>
        <input type="text" name="livro.preco" /></li>

    <li>Data de publicacao: <br/>
      <input type="text" name="livro.dataPublicacao" /></li>
  </ul>
  <input type="submit" value="Salvar" />
</form>
```

Assim podemos criar o método do controller que vai receber o post desse formulário. Se quisermos o livro populado, o parâmetro desse método deve se chamar `livro`, que é o prefixo dos inputs do formulário:

```
@Resource
public class LivrosController {
    public void formulario() {}

    public void salva(Livro livro) {
        Estante estante = new UmaEstanteQualquer();
        estante.guarda(livro);
    }
}
```

Criando a classe UmaEstanteQualquer

O código acima não compila, pois não existe a classe `UmaEstanteQualquer`. Mas é possível rapidamente criá-la usando a sua IDE (por exemplo o Eclipse ou o Netbeans). No caso do Eclipse, com o cursor em `UmaEstanteQualquer`, aperte `Ctrl + 1` e escolha a opção de criar a classe. A classe criada já implementará `Estante` e já vai estar com os métodos da interface implementados, porém em branco.

Por enquanto você pode deixar esses métodos em branco. No próximo capítulo, vamos implementá-los fazendo a comunicação com banco de dados e aprendendo como o VRaptor pode nos ajudar nessa tarefa.

Como o método se chama salva, vai receber a requisição em `/livros/salva`, por isso precisamos colocar isso na action do formulario. Essa URI, no entanto, está sem o *context-path*, então podemos usar a tag `<c:url`, ou usar a variável `${pageContext.request.contextPath}` para acrescentá-lo:

```
<form action="<c:url value="/livros/salva"/>" method="post">

ou

<form action="${pageContext.request.contextPath}/livros/salva"
      method="post">
```

Melhor ainda, se você não quer lembrar qual é a URI de um dos métodos do controller, você pode usar o `linkTo`, com o qual você passa o controller e o método e ele retorna a URI correspondente, já com o *context-path*. Passamos o nome do controller entre colchetes e o nome do método após: `${linkTo[NomeDoController].nomeDoMetodo}`. No nosso caso, ficaria assim:

```
<form action="${linkTo[LivrosController].salva}" method="post">
```

Dessa forma fica mais claro o que vai ser executado: a ação do formulário é um link para o `LivrosController` método salva. Quando o usuário preencher o formulário e clicar em `Salvar`, o VRaptor vai pegar cada um dos valores digitados

nos inputs (com o prefixo `livro`) e preencher uma instância de `Livro` com esses valores. Essa instância será passada para o método salva, com o qual guardamos o livro na `Estante`. Após executar o método `salva`, o VRaptor automaticamente redireciona para a página `WEB-INF/jsp/livros/salva.jsp`, na qual podemos indicar que o livro foi salvo:

```
<h2>Livro adicionado com sucesso!</h2>

<p>Veja aqui a
  <a href="${linkTo[LivrosController].lista}">
    lista de todos os livros
  </a>
</p>
```

Colocamos um link para a lista de todos os livros, assim podemos ver tudo o que foi cadastrado. Agora precisamos criar essa listagem, primeiramente criando o método que colocamos no link:

```
class LivrosController {
    //...
    public void lista() {

    }
}
```

Nesse caso queremos mostrar a lista de todos os livros, que vamos buscar de algum lugar — da `Estante` onde estamos guardando os livros — e queremos deixar essa lista disponível para usar na `jsp`. Como estamos executando um método na requisição, podemos usar o jeito natural do Java, retornando o valor, no caso a lista de livros:

```
public List<Livro> lista() {
    Estante estante = new UmaEstanteQualquer();
    return estante.todosOsLivros();
}
```

Por padrão, o VRaptor deixa o retorno dos métodos do controller disponível na JSP, seguindo outra convenção. Se o retorno do método é um objeto do tipo `Livro`, ele será colocado numa variável chamada `${livro}`, ou seja, nome da classe com a primeira letra em minúsculo.

No caso do retorno ser uma List<Livro>, o nome da variável no JSP será ${livroList}, ou seja, o nome da classe dos elementos da lista com a primeira minúscula, seguido de List.

Podemos criar o lista.jsp dentro de WEB-INF/jsp/livros, que é o jsp padrão para esse método lista:

```
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
    <li>${livro.titulo} - ${livro.descricao}</li>
</c:forEach>
</ul>
```

Por enquanto, para ver essa lista preenchida, precisamos modificar a classe UmaEstanteQualquer para retornar uma lista de livros.

```
@Override
public List<Livro> todosOsLivros() {
  Livro vraptor = new Livro();
  vraptor.setIsbn("123-45");
  vraptor.setTitulo("VRaptor 3");
  vraptor.setDescricao("Um livro sobre VRaptor 3");

  Livro arquitetura = new Livro();
  arquitetura.setIsbn("5678-90");
  arquitetura.setTitulo("Arquitetura");
  arquitetura.setDescricao("Um livro sobre arquitetura");

  return Arrays.asList(vraptor, arquitetura);
}
```

3.3 Complementando o cadastro

Os livros são o carro-chefe do nosso sistema e para poder vendê-los precisamos que as suas informações sejam completas e de qualidade. Já criamos a listagem de livros e o formulário de criação de um livro. Isso não é o bastante para manter nossa livraria funcionando, pois eles podem sofrer alterações ao longo do tempo, principalmente no seu preço, então vamos criar a possibilidade de editar os livros existentes.

Vamos modificar a listagem adicionando um link para permitir a alteração de um livro. Colocaremos um link para a edição em cada livro mostrado:

```
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
    <li>${livro.titulo} - ${livro.descrição}
        <a href="${linkTo[LivrosController].edita}">Modificar</a></li>
</c:forEach>
</ul>
```

Esse link deverá abrir um formulário de edição, que é bem parecido com formulário de criação, mas precisa vir com os dados do livro que estamos querendo modificar já preenchidos. Para isso, precisamos identificá-lo de alguma forma, por exemplo, usando o ISBN. Para passar esse valor para o método `edita`, usamos a forma que estamos acostumados para receber um valor — através de um parâmetro no método:

```
public class LivrosController {

    public void formulario() {}
    public void edita(String isbn) {}
}
```

Como chamamos o parâmetro do método de `isbn`, podemos passar esse parâmetro no link "Modificar", passando um parâmetro na URI com o mesmo nome — `isbn`. Dessa forma, o VRaptor saberá que o valor passado na URI será passado como parâmetro do método:

```
<a href="${linkTo[LivrosController].edita}?isbn=${livro.isbn}">
    Modificar
</a>
```

Assim, se o ISBN do livro é `123-45`, o link gerado será `/livros/edita?isbn=123-45`. O valor `123-45` será passado como parâmetro do método, e podemos usá-lo para buscar o livro desejado na nossa `Estante`.

Para mostrar essa `Estante` na página vamos usar a mesma estratégia da listagem: retornar o livro no método `edita`.

```
public Livro edita(String isbn) {
    Estante estante = new UmaEstanteQualquer();
    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    return livroEncontrado;
}
```

Criando o método buscaPorIsbn

Novamente o código acima não compila e o método `buscaPorIsbn` fica com erro de compilação. Para criá-lo na interface `Estante`, use o `Ctrl + 1` no Eclipse e escolha a opção de criar esse método. Esse atalho funciona toda vez que algo está com erro de compilação por não existir, e é apresentada uma solução rápida para o problema (*Quick Fix*). Experimente usá-lo para fazer a classe `UmaEstanteQualquer` compilar.

Queremos, agora, mostrar os dados desse livro num formulário, assim vamos poder modificá-lo. Poderíamos simplesmente copiar o `formulario.jsp` pra `edita.jsp` e deixar esse formulário preenchido, mas como toda alteração em um formulário vai ter que ser refeita no outro, vamos aproveitar o mesmo arquivo para os dois formulários. Para isso, precisamos modificar o `formulario.jsp` para que sirva tanto para criar um livro novo quanto para editar um já existente.

No resultado do método `edita` podemos usar `${livro.titulo}` para mostrar o título do livro, `${livro.preco}` para mostrar o preço e assim por diante. Para preencher os inputs com esses valores, colocamo-los no atributo `value` de cada um:

```
<form action="${linkTo[LivrosController].salva }" method="post">
  <h2>Formulário de cadastro de livros</h2>
  <ul>
```

```
    <li>Titulo: <br/>
        <input type="text" name="livro.titulo"
                        value="${livro.titulo}"/></li>

    <li>Descricao: <br/>
        <textarea name="livro.descricao">${livro.descricao
        }</textarea></li>

    <li>ISBN: <br/>
        <input type="text" name="livro.isbn"
                        value="${livro.isbn}"/></li>

    <li>Preco: <br/>
        <input type="text" name="livro.preco"
                        value="${livro.preco}"/></li>

    <li>Data de publicacao: <br/>
      <input type="text" name="livro.dataPublicacao"
                      value="${livro.dataPublicacao}"/>
    </li>
  </ul>
  <input type="submit" value="Salvar" />
</form>
```

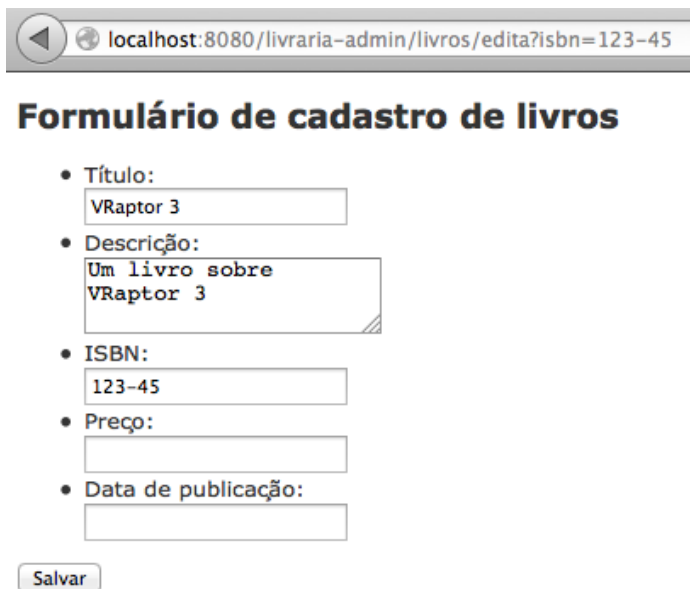
No resultado do método `formulario` não existe um `Livro` retornado, então as expressões `${livro.titulo}`, `${livro.preco}` etc ficarão em branco, que é o esperado no formulário de adição. Agora só falta falar para o VRaptor usar o `formulario.jsp` como resultado do método `edita`. Para modificar o resultado padrão de um método, o VRaptor possui a classe `Result`, que podemos receber como parâmetro do método de ação:

```
public Livro edita(String isbn, Result result) {
    //...
}
```

Falaremos mais sobre o `Result` no capítulo 5, por enquanto só queremos usar a mesma página do método `formulario`, ou seja, o mesmo **resultado** do método `formulario` desse mesmo *controller*. Dizemos isso com o método `of` do `Result`:

```
public Livro edita(String isbn, Result result) {
    Estante estante = new UmaEstanteQualquer();
```

```
    Livro livroEncontrado = estante.buscaPorIsbn(isbn);

    result.of(this).formulario();

    return livroEncontrado;
}
```

Traduzindo, queremos o resultado (`result`) desse controller (`of(this)`) no método formulário (`.formulario()`). Como modificamos o resultado padrão, não podemos mais retornar o `Livro` nesse método, pois o retorno aconteceria após a mudança do resultado. Retornamos o `livroEncontrado` porque queremos incluí-lo na página do formulário, ou seja, incluí-lo no resultado do método, então podemos usar o `Result` para expressar essa intenção com o método `include`:

```
public void edita(String isbn, Result result) {
    Estante estante = new UmaEstanteQualquer();

    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    result.include(livroEncontrado);

    result.of(this).formulario();
}
```

Dessa forma, incluímos o `livroEncontrado` no resultado, da mesma forma que fazíamos no retorno do método, mas **antes** de redirecionar para a página do formulário. Apesar da variável no código java ser `livroEncontrado`, esse objeto passará a `jsp` como `${livro}`, que é o nome da classe com a primeira letra minúscula. E a página usada será a **WEB-INF/jsp/livros/formulario.jsp** e não mais a **edita.jsp**. Nesse caso, só modificamos o jsp a ser utilizado — o método `formulario` **não** será executado.

Para conseguirmos ver o formulário preenchido adequadamente, podemos modificar o método `buscaPorIsbn` da classe `UmaEstanteQualquer`, retornando um dos livros:

```
@Override
public Livro buscaPorIsbn(String isbn) {
  return todosOsLivros().get(0);
}
```

Após modificar os dados do formulário e clicar em "Salvar", o VRaptor vai executar o método `salva`, conforme o que está configurado, e após guardar o livro na estante a página de sucesso é mostrada. Se quisermos continuar cadastrando ou modificando livros, precisamos clicar no link "Ver todos os livros".

Para tornar o processo mais prático, já que conseguimos alterar o resultado padrão, vamos eliminar essa página de sucesso e retornar direto para a listagem de livros, mostrando uma mensagem de sucesso em cima da listagem.

Seguindo a mesma ideia do método `edita`, poderíamos modificar o `salva` para receber o `Result` e mudar o resultado para a listagem:

```
public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

    result.of(this).lista();
}
```

Mas, como foi dito anteriormente, essa mudança de resultado **não** executaria o método `lista` e, portanto, não mostraria a lista de todos os livros — só a página `lista.jsp` sem a variável `${livroList}` que iria ser preenchida pelo método `lista`. Se queremos executar o método, costumamos falar que vamos **redirecionar**

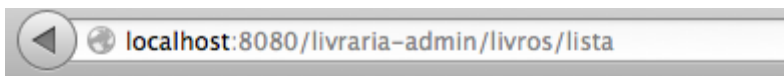
a execução de um método para outro. No nosso caso queremos redirecionar para o método `lista`, então a chamada necessária é o `redirectTo`:

```
public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

    result.redirectTo(this).lista();
}
```

Ou seja, como resultado do método `salva` vamos redirecionar para o método `lista` desse mesmo *controller*. Para indicar que o livro foi salvo, vamos adicionar uma mensagem antes do redirecionamento:

```
public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

    result.include("mensagem", "Livro salvo com sucesso!");
    result.redirectTo(this).lista();
}
```

Nesse caso, estamos incluindo a `String` "Livro salvo com sucesso!" no `result`, e usando a variável `${mensagem}`. Assim podemos modificar nossa listagem para mostrar essa mensagem:

```
<c:if test="${not empty mensagem}">
<p class="mensagem">
  ${mensagem}
</p>
</c:if>
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
    <li>
      ${livro.titulo} - ${livro.descrição}
        <a href="${linkTo[LivrosController].edita}">Modificar</a>
    </li>
</c:forEach>
</ul>
```

Conseguimos construir até aqui as operações mais importantes do cadastro de livros, praticamente só usando as convenções básicas do VRaptor. Criamos URLs para executar o código do `LivrosController`, usando a anotação `@Resource`. Com os jsps no caminho certo, conseguimos linkar para os métodos dos controllers usando o `${linkTo[NomeDoController].metodo}`. Com os inputs de um formulário seguindo a convenção de nomes, conseguimos preencher objetos e usá-los no controller, como vimos no método `salva`. E, por fim, conseguimos alterar o resultado de um método do controller usando o `Result`.

Nos próximos capítulos veremos que o VRaptor nos ajuda muito mais, não somente no controle das requisições, mas também na organização dos componentes da sua aplicação, afinal o desenvolvimento de uma aplicação vai muito além de simples cadastros. Veremos também como integrar o que fizemos com o banco de dados e outros serviços.

Capítulo 4

Organização do código com Injeção de Dependências

4.1 Completando o funcionamento do Controller

No capítulo anterior, os métodos do `LivrosController` foram implementados usando uma classe chamada `UmaEstanteQualquer`, para executar as operações:

```
public List<Livro> lista() {
    Estante estante = new UmaEstanteQualquer();
    return estante.todosOsLivros();
}
```

Isso foi feito porque, neste primeiro momento, para construirmos os métodos do *controller*, bastava apenas qualquer implementação de `Estante` fornecendo a interface necessária. Porém, para termos o sistema funcionando, precisamos definir uma implementação real de `Estante`, que vai guardar os livros para podermos mostrá-los em seguida. São várias as maneiras de fazer isso. Por exemplo, guardando

os livros em arquivos ou usando algum serviço de armazenamento de dados, mas a forma mais comum é usar uma implementação que guarda os livros num **banco de dados**. Vamos, então, mudar a implementação de `Estante` do controller para a `EstanteNoBancoDeDados`.

```
public List<Livro> lista() {
    Estante estante = new EstanteNoBancoDeDados();
    return estante.todosOsLivros();
}
```

Mas para se conectar a um banco de dados e conseguir salvar livros lá, é necessário informar qual é o sistema de banco de dados utilizado, usuário e senha para fazer a conexão e qual é a base de dados, logo poderíamos ter o seguinte código:

```
public List<Livro> lista() {
    Estante estante = new EstanteNoBancoDeDados(BDs.MySQL,
                                "usuario", "senha", "db_livraria");
    return estante.todosOsLivros();
}
```

E se essa estante abriu uma conexão com o banco de dados e eu já terminei de usá-la, eu deveria avisar a estante, senão ela correria o risco de manter a conexão aberta indefinidamente. Então, precisamos colocar o código para fechar essa conexão. Uma `Estante` não precisa ser fechada, mas a implementação do banco de dados sim, então o método pra isso só deveria estar na implementação. Dessa forma, o código ficaria:

```
public List<Livro> lista() {
    Estante estante = null;
    try {
        estante = new EstanteNoBancoDeDados(BDs.MySQL,
                            "usuario", "senha", "db_livraria");
        return estante.todosOsLivros();
    } finally {
        ((EstanteNoBancoDeDados)estante).fechaConexao();
    }
}
```

Repare como o código ficou bem mais complexo que antes, isso usando apenas a `Estante`. Imagine agora esse código espalhado por cada método do seu sistema que depende do banco de dado. A complexidade desse código se dá por uma coisa:

a classe `LivrosController` está **criando** a instância de `Estante` para poder trabalhar. Para isso, o `LivrosController` precisa saber qual é a implementação de `Estante` adequada, quais são as configurações necessárias para criar uma instância dessa implementação e, ao final, saber se precisa sinalizar à instância a hora de fechar os recursos abertos. Esse tipo de código está longe de ser responsabilidade da `LivrosController`, portanto não deveria estar aqui.

Além disso, se precisarmos um dia trocar o banco de dados usado, ou passar a usar um serviço de armazenamento, teríamos que passar por todos os ponto do sistema que usam estantes trocando as implementações. E se quisermos guardar os livros só na memória nas máquinas dos desenvolvedores, mas na máquina de produção usar um banco de dados?

Ao criar uma `EstanteNoBancoDeDados` no `LivrosController` estamos **acoplando** essas duas classes: toda vez que precisarmos alterar a forma de trabalhar da `EstanteNoBancoDeDados` temos que mudar também a `LivrosController`. No capítulo anterior usamos `UmaEstanteQualquer` justamente porque não faz diferença para o `LivrosController` qual é a implementação da `Estante`, só precisa de uma `Estante` pronta pra ser usada. Poderíamos **diminuir o acoplamento** fazendo o `LivrosController` depender apenas do estritamente necessário para ela trabalhar: a **interface** `Estante`.

Alguém terá que criar uma `Estante` e passar para o `LivrosController`, assim o *controller* pode se concentrar naquilo para o que foi feito: controlar as operações com os livros. Mudamos um cenário onde o controller ia atrás de criar e gerenciar a vida da `Estante` para um onde a estante é criada por outra classe e simplesmente passada para o `LivrosController` usar.

Recapitulando, saímos de um *controller* que estava assim:

```
@Resource
public class LivrosController {

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    Estante estante = new UmaEstanteQualquer();
    estante.guarda(livro);

    result.include("mensagem", "Livro salvo com sucesso!");
    result.redirectTo(this).lista();
  }
```

```
  public List<Livro> lista() {
    Estante estante = new UmaEstanteQualquer();
    return estante.todosOsLivros();
  }

  public void edita(String isbn, Result result) {
    Estante estante = new UmaEstanteQualquer();

    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    result.include(livroEncontrado);

    result.of(this).formulario();
  }
}
```

Que, ao trocarmos para a `EstanteNoBancoDeDados`, ficou assim:

```
@Resource
public class LivrosController {

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    Estante estante = null;
    try {
      estante = new EstanteNoBancoDeDados(BDs.MySQL,
          "usuario", "senha", "db_livraria");
      estante.guarda(livro);
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }

    result.redirectTo(this).lista();

  }

  public List<Livro> lista() {
    Estante estante = null;
    try {
      estante = new EstanteNoBancoDeDados(BDs.MySQL,
          "usuario", "senha", "db_livraria");
```

```
        return estante.todosOsLivros();
      } finally {
        ((EstanteNoBancoDeDados) estante).fechaConexao();
      }
    }

    public void edita(String isbn, Result result) {
      Estante estante = null;
      try {
        estante = new EstanteNoBancoDeDados(BDs.MySQL,
            "usuario", "senha", "db_livraria");

        Livro livroEncontrado = estante.buscaPorIsbn(isbn);
        result.include(livroEncontrado);
      } finally {
        ((EstanteNoBancoDeDados) estante).fechaConexao();
      }
      result.of(this).formulario();
    }
}
```

E, para remover a duplicação da criação da estante em todos os métodos, vamos mover isso para o construtor, transformando a `estante` em um atributo da classe:

```
@Resource
public class LivrosController {

  private Estante estante;

  public LivrosController() {
    estante = new EstanteNoBancoDeDados(BDs.MySQL,
        "usuario", "senha", "db_livraria");
  }

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    try {
      estante.guarda(livro);
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }
```

```
    result.redirectTo(this).lista();

  }

  public List<Livro> lista() {
    try {
      return estante.todosOsLivros();
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }
  }

  public void edita(String isbn, Result result) {
    try {
      Livro livroEncontrado = estante.buscaPorIsbn(isbn);
      result.include(livroEncontrado);
    } finally {
      ((EstanteNoBancoDeDados) estante).fechaConexao();
    }
    result.of(this).formulario();
  }
}
```

Mas não queremos que o `LivrosController` se preocupe com a criação da `Estante`, vamos passar a recebê-la, pelo construtor:

```
@Resource
public class LivrosController {

  private Estante estante;

  public LivrosController(Estante estante) {
    this.estante = estante;
  }
  // ...
}
```

E como agora o `LivrosController` não sabe mais qual é a implementação de `Estante` recebida, ele não manda mais a estante fechar a conexão. O código fica:

```
@Resource
public class LivrosController {
```

```
  private Estante estante;

  public LivrosController(Estante estante) {
    this.estante = estante;
  }

  public void formulario() {}

  public void salva(Livro livro, Result result) {
    estante.guarda(livro);

    result.redirectTo(this).lista();
  }

  public List<Livro> lista() {
    return estante.todosOsLivros();
  }

  public void edita(String isbn, Result result) {
    Livro livroEncontrado = estante.buscaPorIsbn(isbn);
    result.include(livroEncontrado);

    result.of(this).formulario();
  }
}
```

Agora o código do controller só tem o que é necessário. Seu código fica bem mais simples e fácil de entender e manter.

Por outro lado, quem criava uma instância de `LivrosController`, antes só fazia um *new*, como em:

```
LivrosController controller = new LivrosController();

controller.lista();
```

Agora, ele precisa também escolher uma implementação de `Estante`, gerenciar essa implementação, e passar para o controller, antes de poder usá-lo:

```
EstanteNoBancoDeDados estante = null;
try {
```

```
  estante = new EstanteNoBancoDeDados(BDs.MySQL,
        "usuario", "senha", "db_livraria");

  LivrosController controller = new LivrosController(estante);

  controller.lista();
} finally {
  estante.fechaConexao();
}
```

Removemos toda essa complexidade do *controller*, que agora está bem mais fácil de se trabalhar, mas transferimos essa complexidade para outra classe, que terá que fazer o trabalho sujo.

4.2 Inversão de Controle: Injeção de Dependências

A prática que vimos anteriormente é chamada de **Inversão de Controle**. Ela consiste em extrair trechos mais voltados à infraestrutura das camadas mais externas da aplicação — por exemplo, nossos controllers. Esses trechos são isolados em classes especializadas, que interagem em uma camada mais interna, onde têm condições de centralizar o gerenciamento das complexidades que antes ficavam espalhadas por todos o sistema.

in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis." (http://en.wikipedia.org/wiki/Inversion_of_control) . PM]

Uma das técnicas para inverter o controle é justamente a que usamos, e ela se chama **Injeção de Dependências**. Nessa técnica, evitamos que a classe controle a criação e o gerenciamento das suas dependências. Em vez disso, declaramos quais são os componentes necessários para o funcionamento de cada classe e confiamos que instâncias funcionais dos componentes serão injetados antes que a classe seja usada. Assim, a responsabilidade de criar e gerenciar os componentes do sistema vai sendo empurrada para camadas inferiores até que a centralizamos num componente especializado que coordenará a injeção das dependências nos lugares certos.

Esse componente especializado é chamado de **container** ou **provedor** (*provider*) de dependências. Dessa forma, cada componente declara quais são as suas dependências, se possível como *interfaces* para ficarmos livres para usar qualquer implementação, e registramos esse componente como uma implementação disponível para a injeção no container.

Ao tentar instanciar o componente, o container buscará cada dependência e, se necessário, criará uma nova instância dessa dependência, que também tem suas próprias dependências e esse processo segue até que todas as dependências sejam resolvidas e, então, o componente requisitado estará criado e pronto para usar.

Uma das filosofias do VRaptor é facilitar e incentivar as melhores práticas de desenvolvimento de software e, para incentivar a **Injeção de Dependências**, o próprio VRaptor possui um container de injeção de dependências embutido. Todos os componentes do sistema que são gerenciados pelo VRaptor podem usar essa técnica e deixar o VRaptor cuidar da resolução das dependências. Para ver como isso funciona, vamos voltar ao controller.

```
@Resource
public class LivrosController {

    private final Estante estante;

    public LivrosController(Estante estante) {
        this.estante = estante;
    }

    // resto do código, substituindo as criações de
    // estantes por this.estante
}
```

Ao recebermos uma `Estante` no construtor, estamos declarando que o `LivrosController` não pode ser criado sem antes receber uma `Estante` preenchida e funcionando, ou seja, declaramos que `Estante` é uma dependência. Como o controller é uma classe gerenciada pelo VRaptor (por causa do `@Resource`), o VRaptor tentará buscar uma implementação de `Estante` candidata a ser injetada. Como `Estante` é uma interface, precisamos indicar qual é a implementação dessa interface que desejamos usar no sistema. Fazemos isso escolhendo a classe que implementa `Estante` mais apropriada e anotando-a com `@Component`:

```
@Component
public EstanteNoBancoDeDados implements Estante {
    //...
}
```

Agora que sabemos que essa implementação usará um banco de dados, precisamos definir como será o acesso a esses dados. Em Java existem várias maneiras de

fazer isso, por exemplo usando *JDBC*, que é a especificação de como nos conectamos ao banco de dados e executamos SQL, ou *Hibernate*, que mapeia os dados de classes Java para tabelas, ou ainda a *JPA* que é a especificação para fazer tal mapeamento.

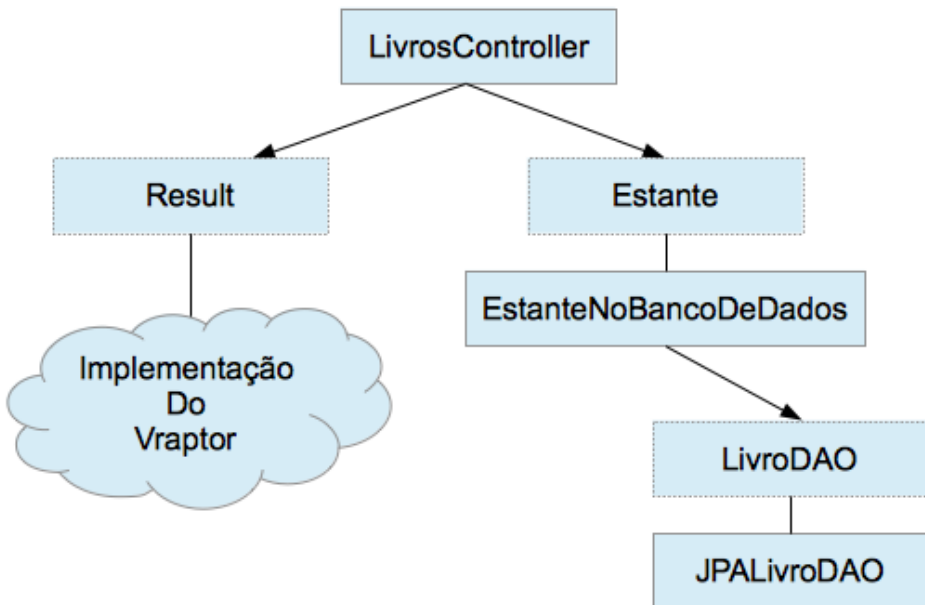
Como a forma de acessar o banco de dados é muito diferente, e às vezes muito extensa, em cada uma dessas possibilidades, não queremos deixar esse tipo de código espalhado pelo sistema, então encapsulamos esse **acesso** aos **dados** num **objeto** que nos dará uma interface independente da ferramenta que usamos para isso. O nome que damos a esse tipo de objeto que acessa dados é **DAO** (Data Access Object).

A `EstanteNoBancoDeDados` precisará acessar os dados de livros no banco, portanto ela depende de um `DAO` de livros — um `LivroDAO`. Para declarar essa dependência fazemos como no controller, recebendo o *DAO* no construtor:

```
@Component
public EstanteNoBancoDeDados implements Estante {

    private final LivroDAO dao;

    public EstanteNoBancoDeDados(LivroDAO dao) {
        this.dao = dao;
    }
    //...
}
```

Ao anotar a `EstanteNoBancoDeDados` com `@Component` ela passa a ser gerenciada pelo VRaptor, que tentará procurar algum componente gerenciado que seja um `LivroDAO`, ou seja, que implemente essa interface. Se formos usar a `JPA` para acesso aos dados, podemos criar uma implementação baseada nisso, e anotar com `@Component` para que ela seja gerenciada pelo VRaptor e possa ser injetada na `EstanteNoBancoDeDados`.

```
@Component
public class JPALivroDAO implements LivroDAO {...}
```

Por usar a JPA, esse DAO precisa de um `EntityManager` para que tudo funcione, logo, recebê-lo-á no construtor. E continuamos indicando os componentes gerenciados até que todas as dependências sejam satisfeitas e então o **Controller** é criado e podemos usá-lo para atender uma requisição para, por exemplo, mostrar a página da lista de todos os produtos.

Num primeiro momento, pode parecer muito trabalhoso declarar todos esses componentes, mas, uma vez criado, um mesmo componente pode ser usado em todas as classes que necessitam dele sem nenhum código adicional, apenas recebendo-o no construtor.

4.3 Implementando a Estante

Agora que sabemos como o VRaptor gerencia as dependências, vamos começar a criar as implementações reais dos nossos componentes. A classe `LivrosController` precisa de uma `Estante` com as seguintes operações:

```
public interface Estante {

    void guarda(Livro livro);
    List<Livro> todosOsLivros();

    Livro buscaPorIsbn(String isbn);
}
```

Como a nossa implementação guardará os livros no banco de dados, criaremos a classe `EstanteNoBancoDeDados`, anotando-a com `@Component` para o VRaptor

usá-la como dependência das outras classes:

```
@Component
public class EstanteNoBancoDeDados implements Estante {
    //...
}
```

Essa classe acessa os dados do livro no banco de dados, portanto podemos usar uma classe que tem essa responsabilidade, ser um `DAO` de livros:

```
@Component
public class EstanteNoBancoDeDados implements Estante {

    private final LivroDAO dao;

    public EstanteNoBanco(LivroDAO dao) {
        this.dao = dao;
    }

    @Override
    public void guarda(Livro livro) {
        this.dao.adiciona(livro);
    }

    @Override
    public List<Livro> todosOsLivros() {
        return this.dao.todos();
    }

    @Override
    public Livro buscaPorIsbn(String isbn) {
        return this.dao.buscaPorIsbn(isbn);
    }
}
```

Diminuindo o acoplamento

Repare que o código do `LivrosController` não precisa ser mudado agora que adicionamos a implementação de `Estante`. Podemos criar essa implementação usando qualquer tecnologia, com qualquer classe do nosso sistema e, ao final, anotar a classe com `@Component` e pronto! Nenhuma das classes que dependem de `Estante` precisará ser modificada.

Conseguimos isso porque o acoplamento do controller é apenas com a **interface** `Estante`, ou seja, é apenas com **o que** a `Estante` consegue fazer e não em **como** é feita a implementação. Por esse motivo, prefira receber interfaces como dependência das suas classes, principalmente quando a implementação depende de uma tecnologia ou biblioteca externa, como FTP ou JPA.

Para essa classe funcionar, precisamos de um `LivroDAO` com a seguinte interface:

```
public interface LivroDAO {
    void adiciona(Livro livro);
    List<Livro> todos();
    Livro buscaPorIsbn(String isbn);
}
```

No nosso sistema, acessaremos o banco de dados com a ajuda da JPA, a especificação do Java para persistência de objetos. Esses objetos persistidos no banco recebem o nome de **entidade**, então, para indicar que um `Livro` tem esse papel, anotamos a classe com `@Entity`.

Como estamos lidando com bancos de dados, ao salvar uma entidade, precisamos de um valor que a identifique de forma única, de modo que podemos recuperar o valor salvo de forma fácil. Por isso é obrigatório que um dos campos da entidade esteja anotado com `@Id`, indicando qual é o identificador da entidade. No caso do livro, temos o ISBN que já é um identificador, então poderíamos usá-lo como `@Id`, mas para conseguirmos distinguir facilmente um livro que já está salvo no banco de um que está sendo criado, vamos criar um atributo id, numérico e automaticamente gerado pelo banco, para atuar como identificador. A classe `Livro`, então, ficaria assim:

```
@Entity
public class Livro {
    @Id @GeneratedValue
    private Long id;

    @Column(unique=true)
    private String isbn;

    private String titulo;
    private String descricao;
    private BigDecimal preco;
    private Calendar dataPublicacao;

    //getters/setters
}
```

Colocando o id no formulário

Para que o formulário continue funcionando para a edição, precisamos acrescentar um campo que represente o id. Como esse campo vai ser automaticamente gerado, não queremos que o usuário altere o id. Por esse motivo, vamos acrescentar um `input` do tipo `hidden` no formulário para guardar o id:

```
<form ....>
   <input type="hidden" name="livro.id" value="${livro.id}" />
   ...
</form>
```

Como o `Livro` é uma entidade, a `JPA` disponibiliza o gerenciador de entidades para realizar as operações de persistência. Esse gerenciador se chama `EntityManager`, e será necessário para que o `LivroDAO`, baseado na JPA, realize o seu trabalho. Ou seja, uma dependência que também será recebida no construtor.

```
@Component
public class JPALivroDAO implements LivroDAO {

  private final EntityManager manager;
```

```
  public JPALivroDAO(EntityManager manager) {
    this.manager = manager;
  }

  @Override
  public void adiciona(Livro livro) {
    if (livro.getId() == null) {
      this.manager.persist(livro);
    } else {
      this.manager.merge(livro);
    }
  }

  @Override
  public List<Livro> todos() {
    return this.manager
            .createQuery("select l from Livro l", Livro.class)
            .getResultList();
  }

  @Override
  public Livro buscaPorIsbn(String isbn) {
    try {
      return this.manager
              .createQuery("select l from Livro l where l.isbn = :isbn",
                          Livro.class)
              .setParameter("isbn", isbn)
              .getSingleResult();
    } catch (NoResultException e) {
      return null;
    }
  }
}
```

Agora nosso `JPALivroDAO` funciona, desde que lhe seja passado um `EntityManager` pronto para ser usado. Mas ainda não registramos nenhum componente que seja um `EntityManager`.

Se tentarmos rodar o sistema do jeito que está, o VRaptor procurará por um `EntityManager` mas não vai encontrar. Nesse caso, acontecerá um erro na aplicação falando que o DAO precisa de um `EntityManager`, mas o VRaptor não sabe

como criá-lo.

Mas se o `EntityManager` é um componente da JPA, como devemos registrá-lo no VRaptor, sendo que não é possível anotá-lo com `@Component`, já que não temos seu código fonte?

```
com.google.inject.CreationException: Guice creation errors:

1) No implementation for javax.persistence.EntityManager was bound.
  while locating javax.persistence.EntityManager
    for parameter 0 at br.com.casadocodigo.livraria.persistencia.JPALivroDAO.<init>(JPALivroDAO.java:15)
  at br.com.caelum.vraptor.ioc.guice.GuiceComponentRegistry.bindToConstructor(GuiceComponentRegistry.java:151)

1 error
```

4.4 Criando objetos complicados - ComponentFactory

O nosso `LivroDAO` necessita de um `EntityManager` para conseguir acessar o banco de dados e, seguindo a injeção de dependências, recebemos um objeto dessa classe no construtor.

Precisamos agora de uma implementação de `EntityManager`, para que o VRaptor consiga injetar essa dependência. Mas essa implementação será feita pela nossa aplicação? Não! Usaremos o Hibernate como implementação da JPA, dessa forma, o VRaptor teria que instanciar uma classe do Hibernate. Podemos colocar `@Component` numa classe do Hibernate?

No caso em que a implementação da nossa dependência não é uma classe da nossa aplicação, não podemos mais usar a estratégia de anotar a implementação com `@Component` para gerenciá-la — temos que usar algo um pouco mais flexível. Além disso, criar um `EntityManager` não é simplesmente dar um `new` em alguma classe determinada. O código para isso é algo parecido com:

```
EntityManagerFactory factory = Persistence
            .createEntityManagerFactory("default");
//...

EntityManager manager = factory.createEntityManager();
```

Para instruir o VRaptor a executar o código acima toda vez que precisar de um `EntityManager`, podemos implementar a interface `ComponentFactory`, que possibilita a criação de um objeto executando um código qualquer. Esse código

deve ser colocado dentro do método `getInstance()` de um `@Component` que implementa essa interface, tipada em `<EntityManager>`:

```
@Component
public class FabricaDeEntityManager
        implements ComponentFactory<EntityManager> {

    @Override
    public EntityManager getInstance() {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("default");

        EntityManager manager = factory.createEntityManager();
        return manager;
    }

}
```

Dessa forma, toda vez que o VRaptor precisar de um `EntityManager` para passar no construtor de alguma classe, ele invocará o método `getInstance()` do nosso `ComponentFactory` e usará o seu resultado. Do jeito que está implementado, cada classe que precisar de um `EntityManager` terá uma cópia diferente — o `getInstance()` sempre retorna um `EntityManager` novo. Se quisermos retornar o mesmo, precisamos criá-lo fora do método, por exemplo, no construtor da classe:

```
@Component
public class FabricaDeEntityManager
        implements ComponentFactory<EntityManager> {

    private final EntityManager manager;

    public FabricaDeEntityManager() {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("default");
        this.manager = factory.createEntityManager();
    }

    @Override
    public EntityManager getInstance() {
        return this.manager;
```

```
    }

}
```

Assim, enquanto estivermos usando a mesma instância da `FabricaDeEntityManager`, o mesmo `EntityManager` será retornado. Mas quanto tempo deverá durar essa instância?

4.5 Tempo de vida dos componentes - Escopo

Quando estamos em um ambiente de injeção de dependências, temos que indicar como os componentes serão criados, para que o nosso *container* possa fazer isso quando necessário. Entretanto, não é só o **como** que importa, precisamos saber também **quando** os componentes serão **criados** e quando eles não serão mais necessários, ou seja, o momento de serem **destruídos**.

Isso é o que chamamos de **escopo** do componente, o tempo em que ele será usado no sistema e após o qual sua instância poderá ser jogada fora.

Durante um escopo, todas as classes que dependerem de um componente receberão a mesma instância deste componente. Como estamos num ambiente *Web*, toda interação com o usuário, como cliques em links ou submissões de formulários, se dá através de **requisições**. Logo um escopo natural é o de **requisição**: todo objeto que precisar ser criado durante uma requisição será destruído ao final dela, ou seja, após a resposta ser devolvida para o usuário.

Sendo assim, toda classe gerenciada pelo VRaptor, como as anotadas com `@Resource` ou `@Component`, é por padrão de escopo de requisição, ou seja, suas instâncias criadas durante a requisição serão destruídas quando ela acabar. Mas existem outros escopos interessantes que podemos usar em qualquer componente, usando as anotações listadas abaixo na classe do componente:

- `@SessionScoped`: escopo de sessão. O objeto será criado por usuário do sistema, atrelado à sua sessão HTTP (`HttpSession`). Ao final da sessão, seja por inatividade ou por chamada explícita, o objeto será destruído.

- `@ApplicationScoped`: escopo de aplicação. Apenas uma instância do componente será criada na aplicação. Também chamado de *singleton*, em alusão ao *Design Pattern* com o mesmo nome.

- `@PrototypeScoped`: escopo de protótipo. Uma instância do componente será criada a cada vez que ele for usado como dependência, ou seja, cada objeto terá uma instância diferente desse componente.
- `@RequestScoped`: escopo de request, o escopo padrão. Cada request terá uma instância diferente desse componente. Você pode usar essa anotação para deixar mais claro o escopo do componente.

Durante esse livro você aprenderá situações em que cada um dos escopos se encaixam melhor e quando usar cada um deles.

Atenção: compatibilidade de escopos

Como alguns escopos são maiores que outros, temos uma restrição importante: um componente não pode ser dependência de um outro de escopo maior. Por exemplo, se um componente é de escopo de requisição, não pode ser dependência de um componente de escopo de sessão ou aplicação, pois ele vai ser destruído antes do objeto que depende dele e o outro objeto ficará num estado inválido.

Se existe um componente `ColetorDeEstatisticas` que é único para toda a aplicação (`@ApplicationScoped`), ele não pode receber uma `Estante` no construtor, pois a `EstanteNoBancoDeDados` é de escopo de requisição (o escopo padrão). Se isso acontecesse, na primeira requisição em que o `ColetorDeEstatisticas` fosse usado, ele iria receber uma `Estante`. Na próxima, ele usaria a mesma `Estante` da última requisição, que já foi jogada fora, pois o seu escopo acabou.

Existe outra particularidade quando estamos trabalhando com uma `ComponentFactory`. O escopo é dado à fábrica e não ao objeto fabricado. O método `getInstance` será chamado sempre, independente do escopo da `ComponentFactory`. Na `FabricaDeEntityManager` movemos o código de criação do `EntityManager` para o construtor justamente por esse motivo:

```
@Component
public class FabricaDeEntityManager
        implements ComponentFactory<EntityManager> {
```

```
    private final EntityManager manager;

    public FabricaDeEntityManager() {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("default");
        this.manager = factory.createEntityManager();
    }

    @Override
    public EntityManager getInstance() {
        return this.manager;
    }

}
```

Instâncias dessa fábrica serão criadas a cada requisição, e cada instância vai criar seu próprio `EntityManager`. Só que cada instância também cria uma `EntityManagerFactory`, que é um objeto de criação cara, que deveria ser único na aplicação inteira, ou seja, seu escopo deveria ser o de aplicação. Para aproveitar a mesma `factory` em todos os requests da aplicação, precisamos gerenciá-la de forma separada. Vamos começar recebendo a `factory` como dependência:

```
@Component
public class FabricaDeEntityManager
        implements ComponentFactory<EntityManager> {

    public FabricaDeEntityManager(EntityManagerFactory factory) {
        this.manager = factory.createEntityManager();
    }
    //...
}
```

Agora precisamos criar a `factory` em outro lugar — outra `ComponentFactory`, com o escopo correto:

```
@Component
@ApplicationScoped
public class FabricaDeEntityManagerFactory
    implements ComponentFactory<EntityManagerFactory> {

    private final EntityManagerFactory factory;
```

```
    public FabricaDeEntityManagerFactory() {
        this.factory = Persistence.createEntityManagerFactory("default");
    }

    @Override
    public EntityManagerFactory getInstance() {
        return this.factory;
    }
}
```

Assim, a fábrica será única na aplicação inteira e, como a `EntityManagerFactory` está sendo criada no construtor, a `factory` também será única na aplicação. Reforçando: mesmo que a fábrica seja `@ApplicationScoped`, o método `getInstance()` será chamado sempre que o VRaptor precisar de um `EntityManagerFactory`.

4.6 Callbacks de ciclo de vida

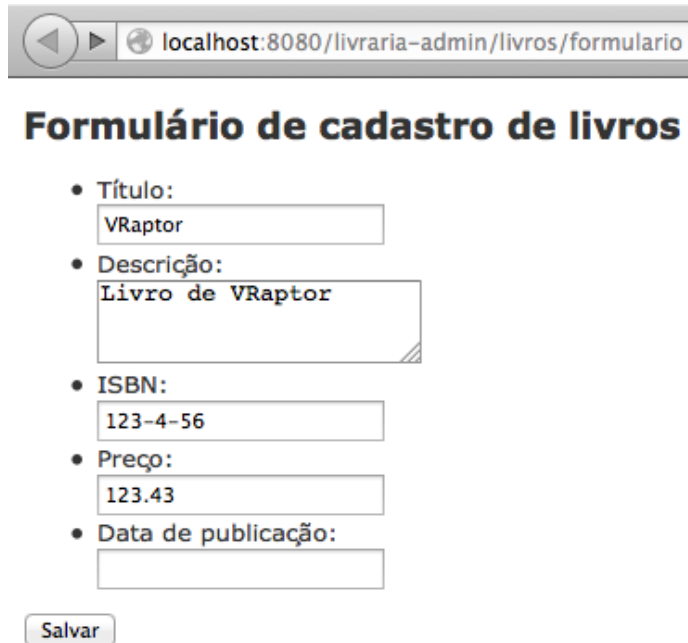
Criamos uma `ComponentFactory` que cria `EntityManager`, mas existe outro fato importante sobre ele: o `EntityManager` abre conexões com o banco de dados e essas conexões **precisam** ser fechadas. Para isso, precisamos chamar o seu método `close()` assim que acabarmos de usá-lo, para evitar o vazamento dessas conexões. Mas **quem** vai chamar esse método? Alguma classe que depende do `EntityManager`? Não podemos fazer desse modo, pois várias outras classes podem estar usando esse mesmo `manager`.

A regra de ouro para recursos que precisam ser fechados/liberados, como conexões, é a seguinte: se uma classe abriu/adquiriu esse recurso, ela é a responsável por fechá-lo/liberá-lo. Por esse motivo, se a `FabricaDeEntityManager` criou o `EntityManager`, ela deveria fechá-lo. Vamos criar um método para isso:

```
@Component
public class FabricaDeEntityManager
        implements ComponentFactory<EntityManager> {

    private final EntityManager manager;
    //...

    public void fechaManager() {
```

```
        this.manager.close();
    }
}
```

O método está criado, mas **quando** ele deve ser chamado? Ele deve ser chamado quando acabarmos de usar o `manager`, ou seja, ao final do escopo da fábrica. Conseguimos fazer isso com o callback `@PreDestroy`. Um método de qualquer componente que estiver anotado com `@PreDestroy` será chamado logo antes do objeto ser destruído, muito útil para liberar os recursos abertos. Esse método **precisa** retornar `void` e não ter argumentos. No nosso caso, queremos chamar o método `fechaManager` ao final do escopo da fábrica, então vamos anotá-lo:

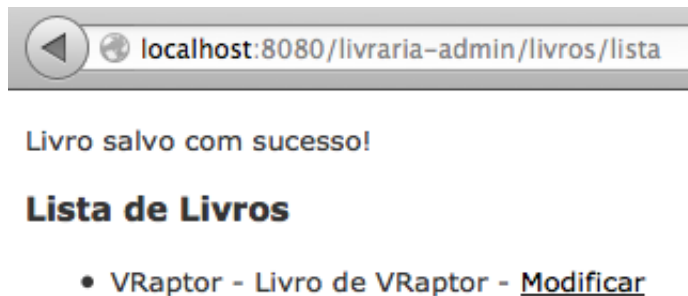
```
@Component
public class FabricaDeEntityManager
        implements ComponentFactory<EntityManager> {

    private final EntityManager manager;
    //...

    @PreDestroy
    public void fechaManager() {
        this.manager.close();
    }
}
```

Pronto, agora temos o nosso criador de `EntityManager` implementado e, como ele era o componente que faltava para o `JPALivroDAO` funcionar, conseguimos subir o servidor e começar cadastrar os livros diretamente no banco de dados. Falta apenas um detalhe: quando estamos trabalhando com bancos de dados só é possível realizar modificações aos dados dentro de uma transação.

Por enquanto, para resolver esse problema vamos alterar o `JPALivroDAO` para usar transações, mas adotaremos uma solução melhor no capítulo 9. O único método que modifica o banco de dados é o método adiciona, portanto vamos controlar a transação dentro dele:

```
@Override
public void adiciona(Livro livro) {
  this.manager.getTransaction().begin();
  this.manager.persist(livro);
  this.manager.getTransaction().commit();
}
```

Formulário de cadastro de livros

- Título: VRaptor
- Descrição: Livro de VRaptor
- ISBN: 123-4-56
- Preço: 123.43
- Data de publicação:

Salvar

Livro salvo com sucesso!

Lista de Livros

- VRaptor - Livro de VRaptor - Modificar

4.7 Outros tipos de injeção de dependência e @PostConstruct

O VRaptor injeta dependências via construtor, a maneira escolhida por exigir o mínimo de configuração: não é possível criar o objeto sem passar todos os argumentos do construtor, logo tudo que está no construtor já é uma dependência da classe. Mas existem outras maneiras de fazer injeção de dependências que são usadas por outras

bibliotecas:

- **Por setter**: bem comum no Spring, você cria um setter para a dependência e indica que esse setter precisa ser invocado, usando uma anotação ou uma configuração em xml.
- **Por atributo**: usado nos EJBs e no CDI, a dependência é injetada diretamente no atributo, via reflection.
- **Por método de inicialização**: um método que recebe de uma vez todas as dependências da classe.

Em todos esses outros tipos, se quisermos executar algum código na inicialização, não podemos usar o construtor: as dependências ainda não foram preenchidas. Nesse caso podemos ter um método anotado com o callback `@PostConstruct`. Esse método será chamado logo após todas as dependências serem preenchidas, quando o objeto está pronto para ser usado.

```
@Inject
public void setDependencia(Dependencia dependencia) {...}

@PostConstruct
public void inicializa() {
    logger.info("Classe inicializada");
    dependencia.fazAlgo();
}
```

O VRaptor não suporta oficialmente esses outros tipos de injeção, mas como ele usa outra biblioteca (Spring, Guice ou Pico Container) para implementar a injeção de dependências, você pode usar a forma da biblioteca escolhida para usar outros tipos de injeção. A anotação `@javax.inject.Inject` funciona em todas essas bibliotecas, pois faz parte da especificação do java para injeção de dependências — basta colocá-la no setter, atributo ou método de inicialização desejado.

Capítulo 5

Tomando o controle dos resultados

Temos agora no nosso sistema o cadastro de livros totalmente funcional, com todas as páginas e os livros sendo salvos e recuperados do banco de dados. Nesse cadastro, vimos que o VRaptor tem a convenção de retornar para uma jsp com o mesmo nome do método executado, na pasta com o nome do controller dentro de `/WEB-INF/jsp`.

Essa é uma convenção muito interessante no caso geral em que o resultado da requisição é uma página HTML, mas nem sempre é isso que queremos e para esses cenários. Vamos precisar sobrescrever essa convenção.

5.1 Redirecionando para outro método do mesmo controller

Um dos resultados possíveis que já vimos é reutilizar a página de outro método, como fizemos no método `edita`:

```
public void edita(String isbn, Result result) {
  //...
  result.of(this).formulario();
}
```

Nessa linha, estamos dizendo para o VRaptor usar o resultado do (`result.of`) método `formulário` desse mesmo objeto (`(this).formulario()`). Consequentemente, o jsp usado será o `/WEB-INF/jsp/livro/formulario.jsp`. Outra mudança da convenção que vimos foi no método `salva` que, ao final da requisição, redireciona para a listagem:

```
public void salva(Livro livro, Result result) {
  //...

  result.redirectTo(this).lista();
}
```

Ou seja, o resultado será um redirecionamento para (`result.redirectTo`) o método `lista` desse mesmo controller (`(this).lista()`).

Essa classe `Result` é o componente do VRaptor responsável pela personalização do resultado final da execução do método do controller. Além de receber no método, podemos recebê-lo no construtor da classe, principalmente se formos usar o `Result` na maioria dos métodos:

```
@Resource
public class LivrosController {
  public final Estante estante;
  public final Result result;

  public LivrosController(Estante estante, Result result) {
    this.estante = estante;
    this.result = result;
  }
  //...
}
```

Através disso, o próprio VRaptor se encarregará de instanciar e disponibilizar o objeto `Result` para nós. Na realidade, o `Result` é um `@Component`, como vimos no capítulo anterior, mas que já vem implementado dentro do próprio VRaptor.

5.2 Disponibilizando vários objetos para as JSPs

Outra convenção que vimos é sobre o retorno do método. Por exemplo, no método `lista`:

```
public List<Livro> lista() {
  return this.estante.todosOsLivros();
}
```

Essa lista de livros será disponibilizada em um atributo da requisição chamado `livroList`, acessível na jsp por `${livroList}`. Mas e se, além dessa lista de livros, quisermos acrescentar também o livro em promoção do dia? Não podemos simplesmente retornar dois objetos, pois isso não é válido em java, então precisamos novamente do `Result` para alterar essa convenção, com o método `include`. Com ele, podemos adicionar quantos objetos forem necessários, e dar nomes diferentes da convenção para eles também:

```
public List<Livro> lista() {
  this.result.include("promocao", this.estante.promocaoDoDia());
  return this.estante.todosOsLivros();
}
```

Para não misturar as convenções, podemos usar o `result` para a lista de livros também:

```
public void lista() {
  this.result.include("promocao", this.estante.promocaoDoDia());
  this.result.include("livros", this.estante.todosOsLivros());
}
```

Dessa forma, a lista de livros não ficará mais acessível por `${livroList}`, mas apenas por `${livros}`. Uma sobrecarga desse método não recebe como parâmetro o nome, como foi usado no método `edita` do controller:

```
public void edita(String isbn) {
  Livro livroEncontrado = this.estante.buscaPorIsbn(isbn);
  result.include(livroEncontrado);

  result.of(this).formulario();
}
```

Nesse caso, o VRaptor usará a convenção para nomes: nome da classe com a primeira letra minúscula. Ou seja, o atributo se chamará `livro`. Essa convenção não funcionará, no entanto, se tentarmos usar uma lista:

```
List<Livro> livros = this.estante.todosOsLivros();
result.include(livros);
```

Não é possível descobrir o tipo genérico da variável `livros`, pois o Java o apaga em tempo de execução, logo a variável aqui seria apenas "list" ao invés de "livroList". Para dar um nome melhor devemos usar a versão do método `include` que recebe uma string:

```
List<Livro> livros = this.estante.todosOsLivros();
result.include("livros", livros);
```

5.3 Mais sobre redirecionamentos

Vimos como redirecionar para outros métodos do mesmo controller mas isso não resolve todos os nossos problemas. Por exemplo, se estivermos no `LivrosController`, invocando um método chamado `comprar` que, ao final da compra, redireciona para a página inicial do sistema. Para isso, podemos usar o redirecionamento que recebe a classe do controller em vez de `this`:

```
public void comprar(Livro livro) {
  //todo o processo de comprar o livro
  result.redirectTo(HomeController.class).paginaInicial();
}
```

Isso funciona para todos os três tipos de redirecionamento que o VRaptor suporta e que podem ser usados ao final do método do *controller*. Esses redirecionamentos são:

- `result.of(this).metodo()`
  ou
  `result.of(UmController.class).metodo()`

  a página `jsp` do método indicado será renderizada, sem executar o método. Útil quando queremos compartilhar a mesma jsp entre dois ou mais métodos.

- `result.forwardTo(this).metodo()` ou `result.forwardTo(UmController.class).metodo()`: executa o método indicado até o final e usa o seu resultado. Esse redirecionamento é transparente para o usuário, pois a URL que permanece no browser é a do primeiro método chamado. É um redirecionamento do tipo *FORWARD*, que é executado do lado do servidor.
- `result.redirectTo(this).metodo()` ou `result.redirectTo(UmController.class).metodo`: redireciona para o método indicado, do lado do cliente, ou seja, a requisição volta para o browser e a URL é trocada para a do novo método. É um redirecionamento do tipo *REDIRECT*, que é executado do lado do cliente.

Redirecionamento no cliente versus Redirecionamento no servidor

Os redirecionamentos do tipo *REDIRECT*, ou seja, do lado do cliente voltam para o browser e executam uma nova requisição limpa, bastante recomendado quando queremos usar como resultado de um método (por exemplo, o método `salva` do controller) outro método (por exemplo, o método `lista`). Se o usuário recarregar a página, a requisição irá direto para o método `lista`, não passará mais pelo método `salva`, assim o navegador não pedirá para ele submeter o formulário novamente.

Já os redirecionamentos do tipo *FORWARD*, isto é, do lado do servidor, executam o segundo método dentro da mesma requisição, reaproveitando todo o estado dessa requisição. Deve ser usado quando quisermos executar as duas lógicas na mesma requisição e quando, ao recarregar a página, quisermos executar as duas lógicas novamente.

5.4 Outros tipos de resultado

Acabamos de ver o uso básico do `Result`, para as operações mais comuns do desenvolvimento de uma aplicação web. No entanto, existem outros tipos de mudanças de resultado que são bastante úteis em determinadas situações. Por exemplo, se tomarmos o método `edita` do `LivrosController`:

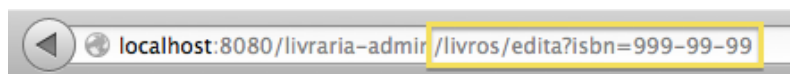
```
public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  result.include(livroEncontrado);

  result.of(this).formulario();
}
```

O que fazer se for passado um `isbn` que não pertence a nenhum livro? Poderíamos criar uma página diferente pra isso e redirecionar para ela:

```
public void naoEncontrado() {}

public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  if (livroEncontrado == null) {
    result.forwardTo(this).naoEncontrado();
  } else {
    result.include(livroEncontrado);
    result.of(this).formulario();
  }
}
```

Mas, em aplicações web, já existe uma página padrão para redirecionarmos quando tentamos acessar algo que não existe: a página 404. Se quisermos redirecionar para essa página, podemos usar o método `notFound` do `Result`. Assim será usada a página 404 configurada no sistema.

```
public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  if (livroEncontrado == null) {
    result.notFound();
  } else {
    result.include(livroEncontrado);
    result.of(this).formulario();
  }
}
```

Formulário de cadastro de livros

- Título:
  VRaptor
- Descrição:
  Livro de VRaptor
- ISBN:
  123-4-56
- Preço:
  123.43
- Data de publicação:

Salvar

Customizando as páginas de erro

É possível customizar as páginas de erro 404 ou 500 padrão do site, colocando no arquivo web.xml:

```
<error-page>
  <error-code>404</error-code>
  <location>/404.jsp</location>
</error-page>
```

Desse modo, podemos mostrar uma página de 404 com a cara do sistema, ao invés da página padrão do tomcat ou do servidor escolhido.

Além desse método, existe o `use` do Result, que nos permite usar diversos tipos de resultado, da seguinte forma:

```
result.use(umTipoDeResultado).configuracaoDesseResultado();
```

Os tipos de resultado que já vêm implementados no VRaptor estão disponíveis através de métodos estáticos na classe `Results`. Esses resultados são:

- `Results.http()`: possibilita a alteração das partes HTTP da resposta, como *status code*, *headers*, ou até mesmo retornar um corpo da requisição. Por exemplo:

  ```
  result.use(Results.http()).body("Deu tudo certo");
  ```

  Assim, ao invés de ir para uma JSP, a resposta da requisição será o texto "Deu tudo certo".

- `Results.status()`: possibilita a alteração do *status code* da requisição. Por exemplo:

  ```
  result.use(Results.status())
        .forbidden("Você não está autorizado a acessar esse conteúdo");
  ```

- `Results.json()`: retorna os dados de um objeto serializados em JSON. Por exemplo:

```
result.use(Results.json()).from(livro).serialize();
```

Nesse caso a resposta será algo como:

```
{ "livro": {
    "titulo": "VRaptor 3",
    "descricao": "Um livro legal sobre VRaptor",
    "isbn": "12345-6"
} }
```

- `Results.jsonp()`: semelhante ao resultado de JSON, mas com a opção de passar um callback de JSONP.
- `Results.xml()`: semelhante ao resultado de JSON, mas serializa o objeto em XML.
- `Results.representation()`: tenta decidir, de acordo com o que veio na requisição, se o o objeto passado será serializado em XML, JSON ou se será redirecionado para a JSP. O uso é similar ao resultado de XML e de JSON.
- `Results.nothing()`: retorna uma resposta vazia. Existe um atalho para isso no próprio result:

```
result.use(Results.nothing());
//ou
result.nothing();
```

- `Results.referer()`: possibilita o redirecionamento de volta para a página que gerou a requisição. Esse resultado usa o *Header Referer* da requisição, que em geral é enviado pelos navegadores, com a *URL* da página atual. No entanto, esse header não é obrigatório e, se não estiver presente na requisição, esse resultado gerará um erro.
- `Results.page()`: agrupa os redirecionamentos para páginas, aqueles que não executam métodos de controllers. Por exemplo:

```
result.use(Results.page()).of(UmController.class).metodo();
// que é o mesmo que:
result.of(UmController.class).metodo();
```

- `Results.logic()`: agrupa os redirecionamentos para métodos de um controller. Por exemplo:

```
result.use(Results.logic()).redirectTo(UmController.class).metodo();
// que é o mesmo que:
result.redirectTo(UmController.class).metodo();
```

Todos esses resultados possuem mais opções, você pode explorá-las usando o *auto-complete* da sua IDE. No caso do Eclipse, você pode fazer, por exemplo:

```
result.use(Results.status()).<Ctrl + Espaço>
```

E você consegue ver a lista de métodos que esse resultado disponibiliza.

Capítulo 6

Validando o seu domínio

Nosso sistema possui um cadastro de livros, com todos os dados editáveis no formulário implementado pelo `LivrosController`. No entanto, não podemos vender qualquer livro cadastrado nesse formulário. No momento é possível cadastrar um livro sem título ou sem preço, que não pode ser colocado à venda no site.

Poderíamos ficar em todas as partes do sistema verificando se os dados do livro (ou qualquer outro modelo do sistema) estão corretos antes de usá-los, mas isso deixa o sistema bastante complicado, cheio de `if`s de 'segurança'. O melhor a fazer é simplesmente não guardar um `Livro` se ele não satisfizer algumas restrições consideradas necessárias para o `Livro` ser usado no sistema. Assim, no resto do código não precisamos nos preocupar em checar os dados do livro: se um livro chegou para nós, ele está pronto para ser usado.

Para implementar essas restrições, por exemplo a obrigatoriedade do título do livro, vamos modificar o método `salva` do nosso `LivrosController`:

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
```

```
    //não deixa salvar!
  }
  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

O que significa não deixar salvar? Esse método `salva` é invocado a partir do formulário de cadastro do `Livro` e, se existe algum campo inválido, precisamos avisar para a pessoa que está preenchendo o formulário o que está errado. Voltando para o método:

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
    //volta para o formulário dizendo que o título é obrigatório
  }
  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

Se, no entanto, existirem mais validações:

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
    //volta para o formulário dizendo que o título é obrigatório
  }
  if (livro.getPreco() == null
        || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
    //volta para o formulário dizendo que o preço é obrigatório e
    //deve ser positivo
  }
  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

Não podemos voltar para o formulário a cada campo que está errado, senão o usuário vai preencher o formulário, clicar em "Salvar" e ver que o título está inválido; então ele arruma o título e clica novamente em "Salvar" e a página fala que o preço está inválido; e assim por diante para cada campo do livro. Melhor seria acumular

todos os erros que acontecerem e mostrar todos de uma vez no formulário, se houver algum erro para ser mostrado. Se não existirem erros podemos guardar o livro e continuar com o processo.

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
    //adiciona o erro "título é obrigatório"
  }
  if (livro.getPreco() == null
        || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
    //adiciona o erro "preço é obrigatório e deve ser positivo"
  }
  if (houverem erros) {
    //volta para o formulário mostrando os erros
  }
  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

Poderíamos controlar tudo isso manualmente, somente usando o `result` para mudar o fluxo, mas como essa tarefa de validar se o objeto a ser salvo é bem comum, existe um componente do VRaptor especializado em executar essas validações: o `Validator`. Nele, podemos adicionar mensagens de erro de validação e, se houver erros, redirecionar a requisição de volta para o formulário, ou seja, exatamente o que a gente queria. Para ter acesso a esse componente, recebemo-lo no construtor:

```
@Resource
public class LivrosController {

  private Estante estante;
  private Result result;
  private Validator validator;

  public LivrosController(Estante estante, Result result,
                          Validator validator) {
    this.estante = estante;
    this.result = result;
    this.validator = validator;
  }
}
```

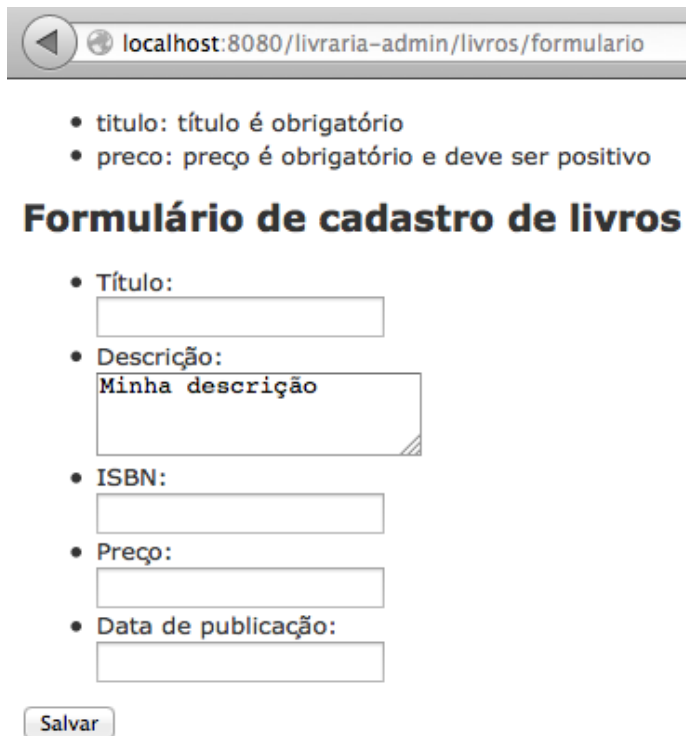
Para adicionar uma validação, usamos o método add do Validator, passando uma Message. Uma das mensagens possíveis é a ValidationMessage, com a qual passamos o texto da mensagem e o campo onde ela ocorreu:

```
if (livro.getTitulo() == null) {
  validator.add(new ValidationMessage("título é obrigatório", "titulo"));
}
if (livro.getPreco() == null
      || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
  validator.add(new ValidationMessage("preço é obrigatório " +
                                      "e deve ser positivo", "preco"));
}
```

E para redirecionar para o formulário, fazemos algo bem parecido com os redirecionamentos do Result. No Validator falamos que, se houver erros, queremos redirecionar para o formulário:

```
validator.onErrorRedirectTo(this).formulario();
```

Isso segue a mesma lógica do Result: estamos redirecionando para esse mesmo controller, no método formulario. Ao voltar para o formulário, todos os erros que ocorreram vão estar disponíveis na variável ${errors}. Para mostrar os erros no formulário, podemos fazer:

```
<ul class="errors">
  <c:forEach items="${errors}" var="error">
    <li>
      <!-- o campo em que ocorreu o erro, ou o tipo do erro -->
      ${error.category}:

      <!-- a mensagem de erro de validação -->
      ${error.message}
    </li>
  </c:forEach>
</ul>
```

O nosso método salva ficaria assim:

```
public void salva(Livro livro) {
  if (livro.getTitulo() == null) {
    validator.add(
      new ValidationMessage("título é obrigatório", "titulo")
```

```
    );
  }
  if (livro.getPreco() == null
        || livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
    validator.add(new ValidationMessage("preço é obrigatório " +
                                  "e deve ser positivo", "preco"));
  }
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

Nada que está abaixo da linha do `onErrorRedirectTo` será executado caso exista algum erro de validação.

- titulo: título é obrigatório
- preco: preço é obrigatório e deve ser positivo

Formulário de cadastro de livros

- Título:
- Descrição: Minha descrição
- ISBN:
- Preço:
- Data de publicação:

Salvar

6.1 Internacionalização das mensagens

Em Livro, temos título e preço como campos obrigatórios. Podemos dizer também que o ISBN é obrigatório, já que é o nosso identificador do livro. Se formos parar pra pensar, cada classe que formos salvar no banco vai ter alguns campos obrigatórios e, em todos eles, vamos fazer uma validação com a mensagem bem parecida:

```
if (livro.getTitulo() == null) {
  validator.add(new ValidationMessage("título é obrigatório", "titulo");
}
```

O que acontece agora se for preciso alterar a mensagem dos campos obrigatórios de "campo é obrigatório" para "campo deve ser preenchido"? Vamos ter que passar pelo sistema todo mudando as mensagens de campo obrigatório. E se precisarmos mostrar as mensagens ora em português, ora em espanhol e ora em inglês? O processo de geração das mensagens ficaria bem complicado e espalhado pelo sistema.

Para resolver esse tipo de problema, o Java possui uma classe chamada ResourceBundle, que foi pensada para separar esse tipo de mensagens do meio do código, além de possibilitar a internacionalização (*i18n*) dessas mensagens. Para fazer isso, precisamos criar um conjunto de arquivos .properties, um para cada língua que formos suportar no sistema.

```
messages.properties       => as mensagens na língua padrão
messages_en.properties    => as mensagens em inglês
messages_es.properties    => as mensagens em espanhol
messages_pt_BR.properties => as mensagens em português do Brasil.
```

Se usarmos os arquivos exatamente nesse padrão, começando com messages e colocando-os no *classpath*, o VRaptor possibilita usar as mensagens desse bundle, com a classe I18nMessage. Nela, passamos primeiro o campo que ocorreu o erro e depois a chave da mensagem:

```
if (livro.getTitulo() == null) {
  validator.add(new I18nMessage("titulo", "campo.obrigatorio"));
}
if (livro.getPreco() == null) {
  validator.add(new I18nMessage("preco", "campo.obrigatorio"));
}
if (livro.getIsbn() == null) {
  validator.add(new I18nMessage("isbn", "campo.obrigatorio"));
}
```

E no arquivo `messages.properties` (ou em alguma das línguas):

```
campo.obrigatorio = deve ser preenchido
```

Podemos, ainda, passar parâmetros para a mensagem, usando os próximos argumentos do construtor do `I18nMessage` e usando `{0}`, `{1}`, `{2}` etc no arquivo de mensagens, representando cada um dos parâmetros adicionais, na ordem dos argumentos.

```
if (livro.getTitulo() == null) {
  validator.add(
    new I18nMessage("titulo", "campo.obrigatorio", "título"));
}

if (livro.getPreco() == null) {
  validator.add(
    new I18nMessage("preco", "campo.obrigatorio", "preco"));
} else if (livro.getPreco().compareTo(BigDecimal.ZERO) < 0) {
  validator.add(
    new I18nMessage("preco", "campo.maior.que", "preço", 0));
}

if (livro.getIsbn() == null) {
  validator.add(new I18nMessage("isbn", "campo.obrigatorio", "isbn"));
}
```

messages.properties:

```
campo.obrigatorio = {0} deve ser preenchido
campo.maior.que = {0} deve ser maior que {1}
```

Assim, as mensagens, se forem geradas, ficariam:

```
título deve ser preenchido
preço deve ser preenchido
preço deve ser maior que 0
isbn deve ser preenchido
```

Por outro lado, se digitarmos um texto qualquer nos campos "Preço" e "Data de publicação", receberemos as mensagens:

```
preco: ???is_not_a_valid_number???
dataPublicacao: ???is_not_a_valid_date???
```

Esses erros são adicionados automaticamente pelo VRaptor se o valor mandado na requisição não puder ser convertido para o tipo necessário. No nosso caso, converter um texto qualquer para `BigDecimal` ou `Calendar`. O texto que está entre `???` é a chave de i18n que podemos usar para gerar a mensagem desses erros. Podemos colocar uma mensagem com essa chave no messages.properties, usando `{0}` para incluir o valor inválido:

```
is_not_a_valid_number = "{0}" não é um número válido
is_not_a_valid_date = "{0}" não é uma data válida
```

E receber as mensagens:

```
preco: "dez reais" não é um número válido
dataPublicacao: "hoje a noite" não é uma data válida
```

Esses erros são adicionados antes do controller ser executado e, caso você não tenha usado o `validator` para dizer para onde ir em caso de erro, receberá a exception:

```
There are validation errors and you forgot to specify where to go.
Please add in your method something like:

validator.onErrorUse(page()).of(AnyController.class).anyMethod();

or any view that you like.
If you didn't add any validation error, it is possible that a
conversion error had happened.
```

6.2 Validação fluente

Outra forma de usar as mensagens internacionalizadas, fugindo um pouco dos ifs, é usando a validação fluente do VRaptor. Nela, declaramos o que queremos que seja verdade, e caso seja falso, o erro de validação é adicionado. Nesse tipo de validação, usamos a classe `Validations`, onde podemos declarar várias validações, e passamos para o método `checking` do `validator`.

```
validator.checking(new Validations() {{
  // queremos que o titulo não seja null. A ordem dos
  // próximos parâmetros é a mesma do construtor de I18nMessage
  that(livro.getTitulo() != null, "titulo", "campo.obrigatorio",
```

```
      "título");

  // esse método that retorna o resultado da condição,
  // assim podemos executar condicionalmente a validação (não podemos
  // checar se o preço é maior que zero se ele for nulo)
  if (that(livro.getPreco() != null, "preco", "campo.obrigatorio",
          "preco"))

    that(livro.getPreco().compareTo(BigDecimal.ZERO) > 0,
          "preco", "campo.maior.que", "preco", 0);

  that(livro.getIsbn() != null, "isbn", "campo.obrigatorio", "isbn");
}});
```

Repare que aqui colocamos a condição que queremos que seja verdadeira, diferentemente da versão com `ifs`.

As duas chaves `{{ }}` são necessárias nesse código, pois estamos criando uma classe anônima filha de `Validations`, com as validações declaradas em sua inicialização. As chaves externas declaram a classe anônima, e as internas declaram um bloco de inicialização, que é executado antes do construtor da classe. Outro ponto é que, se você está usando Java 6 ou 5, as variáveis usadas dentro da `Validations` precisam ser declaradas como `final`. O resultado final do método `salva`, usando essa forma de validação, é o seguinte:

```
public void salva(final Livro livro) {
  validator.checking(new Validations() {{
    that(livro.getTitulo() != null, "titulo", "campo.obrigatorio",
         "título");

    if (that(livro.getPreco() != null, "preco", "campo.obrigatorio",
            "preco"))
      that(livro.getPreco().compareTo(BigDecimal.ZERO) > 0,
          "preco", "campo.maior.que", "preco", 0);

    that(livro.getIsbn() != null, "isbn", "campo.obrigatorio", "isbn");
  }});
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);
```

```
  result.redirectTo(this).lista();
}
```

Repare que sempre é necessário especificar o que fazer caso haja erros e nesse caso redirecionamos para o `formulario`.

Para usar essa forma de validação, é necessário incluir nas bibliotecas da aplicação o Hamcrest. No nosso caso, adicionando ao ivy.xml:

```
<dependency org="org.hamcrest" name="hamcrest-core" rev="1.2"
            conf="default" />
```

E rodando o comando "ant resolve".

Saiba mais: validações poderosas usando Hamcrest

O Hamcrest é uma biblioteca para fazermos *matching* de objetos, muito usada para testes de unidade, para gerarmos asserções mais legíveis em linguagem natural. No caso da validação fluente do VRaptor, podemos compor as checagens do Hamcrest no método `that`:

```
import static org.hamcrest.Matchers.*;
//...
validator.checking(new Validations() {{
  that(livro.getTitulo(), is(notNullValue()));

  that(livro.getPreco(), is(allOf(
      notNullValue(),
      greaterThan(BigDecimal.ZERO)
  )));

  that(livro.getIsbn(), is(notNullValue()));
}});
```

Nesse caso, a mensagem gerada é uma mensagem em linguagem natural, mas em inglês, correspondente às validações usadas. No entanto, é possível mudar as mensagens de validação passando mais parâmetros:

```
that(livro.getTitulo(), is(notNullValue()),
      "titulo", "campo.obrigatorio");
```

Mais informações sobre os matchers possíveis: https://code.google.com/p/hamcrest/wiki/Tutorial

Para usar os matchers do hamcrest no projeto, é necessário adicionar a dependência do `hamcrest-library`:

```
<dependency org="org.hamcrest" name="hamcrest-library"
        rev="1.2" conf="default" />
```

6.3 Organizando melhor as validações com o Bean Validations

Embora o `Validator` do VRaptor ajude bastante, acabamos repetindo bastante código ao declarar validações das maneiras vistas acima. Por exemplo, toda vez que um campo for obrigatório, teremos as linhas:

```
if (objeto.getCampo() == null)
  validator.add(new I18nMessage("campo", "campo.obrigatorio", "campo"));

//ou
that(objeto.getCampo() != null, "campo", "campo.obrigatorio", "campo");
```

É a mesma coisa para outros tipos de validação, como ver se um número é maior que zero, ver se uma string tem no máximo 10 caracteres, ver se uma data está no passado etc. Essas validações estão presentes em quase todo tipo de objeto que vai ser salvo no banco de dados. Por esse motivo, ao invés de ficar repetindo esses `ifs`, poderíamos falar que um determinado campo é obrigatório, e alguém se encarregar de fazer o `if` para nós. Também é o mesmo para um campo que deve ser maior que zero, ou um campo que tem que estar no passado.

Para resolver esse problema, existe uma especificação do Java chamada **Bean Validations**. Com ela, usamos anotações em cima dos campos, para declarar as validações a serem aplicadas. No caso do `Livro` as validações seriam:

```
@Entity
public class Livro {
  @Id @GeneratedValue
  private Long id;

  @NotEmpty
  private String isbn;

  @NotEmpty
```

```
  private String titulo;

  @NotNull @DecimalMin("0.0")
  private BigDecimal preco;

  @Past
  private Calendar dataPublicacao;

  private String descricao;
  //...
}
```

E os `ifs` para verificar se o campo está válido e adicionar a mensagem de validação serão feitos automaticamente. Tudo o que temos que fazer é indicar ao VRaptor que queremos validar o `Livro`, com o método `validate` do `validator`:

```
public void salva(Livro livro) {
  validator.validate(livro);
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

Exemplos de anotações de validação do Bean Validations

A especificação Bean Validations possui algumas anotações já padronizadas para fazermos as validações, mas cada implementação está livre para adicionar novas anotações e é possível criarmos novas anotações na nossa própria aplicação. Se usarmos o Hibernate Validator, algumas das anotações possíveis são:

- Para objetos em geral: `@Null` e `@NotNull`, para garantir que um objeto seja ou não nulo, respectivamente
- Para strings: `@Size(min=0, max=20)`, para validar o tamanho, `@Pattern(regexp="[0-9]*")` para casar padrões, `@Email` e `@CreditCardNumber`, para garantir que a `String` representa um email ou um número de cartão de crédito.
- Para números: `@Min(2)`, `@Max(30)` e `@Range(min=1, max=40)`, para definir limites para um número inteiro, `@DecimalMin("0.01")` e `@DecimalMax("999.99")`, para definir limites para um número decimal e `@Digits(integer=6, fraction=2)` para definir a precisão da parte inteira e da fracionária de um número decimal.
- Para datas: `@Past` e `@Future`, para datas no passado ou no futuro.
- Para booleans: `@AssertTrue` e `@AssertFalse`, para garantir que o boolean é verdadeiro ou falso.

Ainda é possível usar bibliotecas de terceiros, como por exemplo o Caelum Stella, que nos dá validações brasileiras, como o `@CPF` e o `@CNPJ`.

Assim, podemos concentrar a maioria das validações do nosso modelo no próprio modelo. Mas nem sempre é possível usar uma validação do `Bean validations`, como, por exemplo, se quisermos saber se já existe um livro salvo

no banco com o mesmo ISBN. Nesse caso precisamos nos conectar ao banco e fazer uma consulta por livros com um determinado ISBN, mas o nosso modelo `Livro` não possui esse acesso. Logo, devemos fazer essa validação usando as formas vistas anteriormente.

As mensagens de validação geradas com o Bean Validations são em inglês. Se quisermos sobrescrever a mensagem de validação, temos três opções:

- Por mensagem direta:

  ```
  @NotNull(message = "Título precisa ser preenchido")
  private String titulo;
  ```

- Por chave de validação

  ```
  @NotNull(message = "{campo.obrigatorio}")
  private String titulo;
  ```

  Essa chave precisa estar no bundle `ValidationMessages.properties`.

- Genericamente para um tipo de anotação de validação:

  ```
  //Se o import é esse:
  import javax.validation.constraints.DecimalMin;
  ```

  Podemos adicionar no ValidationMessages.properties:

  ```
  javax.validation.constraints.DecimalMin = Deve ser maior que {value}
  ```

  É possível interpolar na mensagem os valores que estão dentro da anotação de validação. Pudemos usar o `{value}` pois usamos na anotação:

  ```
  @DecimalMin("0.0")
  //que é equivalente a
  @DecimalMin(value="0.0")
  ```

6.4 Boas práticas de validação

O Bean Validations é ótimo para implementar a maioria das validações, portanto, idealmente um método do controller que salva algo no banco deveria ser algo como:

```
public void salva(Livro livro) {
  validator.validate(livro);
  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

Ou seja: manda validar o objeto; retorna para o formulário em caso de erro; estando tudo ok, salva o objeto; redireciona para alguma página de sucesso. Um método simples de entender e de dar manutenção. Qualquer validação adicional pode ser feita diretamente no `Livro` com as anotações do Bean Validations sem precisar alterar o controller.

Mas isso não vai resolver todas as validações possíveis, pois só conseguimos validar facilmente os dados de algum dos atributos do `Livro`, então temos que executar a validação manualmente:

```
public void salva(Livro livro) {
  validator.validate(livro);

  if (estante.jaExisteNoBanco(livro.getIsbn())) {
    validator.add(new I18nMessage("isbn", "isbn.duplicado"));
  }

  validator.onErrorRedirectTo(this).formulario();

  estante.guarda(livro);

  result.redirectTo(this).lista();
}
```

Quanto mais validações adicionamos nos métodos dos controllers, mais difícil é entender o que está acontecendo. Para manter o controller o mais simples possível, podemos criar novos validators, por exemplo, o `LivroValidator`, que fará as validações específicas do livro.

```
@Component
public class LivroValidator {
  private Validator validator;
  private Estante estante;
```

```
  private Editoras editoras;

  public LivroValidator(Validator validator,
          Estante estante, Editoras editoras) {
    this.validator = validator;
    this.estante = estante;
    this.editoras = editoras;
  }

  public void validate(Livro livro) {
    validator.validate(livro);

    if (estante.existeLivroComTitulo(livro.getTitulo())) {
      validator.add(new I18nMessage("titulo", "ja.existe"));
    }

    if (editoras.concorrentes().contains(livro.getEditora())) {
      validator.add(new I18nMessage("editora",
            "nao.pode.ser.editora.concorrente"))
    }
  }

  //gere os delegate methods do Validator usando a sua IDE!
  public <T> T onErrorRedirectTo(T controller) {
    return validator.onErrorRedirectTo(controller);
  }
}
```

Dessa forma, nosso método do controller volta a ser simples, bastando receber um `LivroValidator` ao invés do `Validator` do VRaptor.

```
@Resource
public class LivrosController {
  private LivroValidator validator;

  public LivrosController(/*...,*/ LivroValidator validator) {
    //...
    this.validator = validator;
  }

  public void salva(Livro livro) {
    validator.validate(livro);
```

```
    validator.onErrorRedirectTo(this).formulario();

    estante.guarda(livro);

    result.redirectTo(this).lista();
  }
}
```

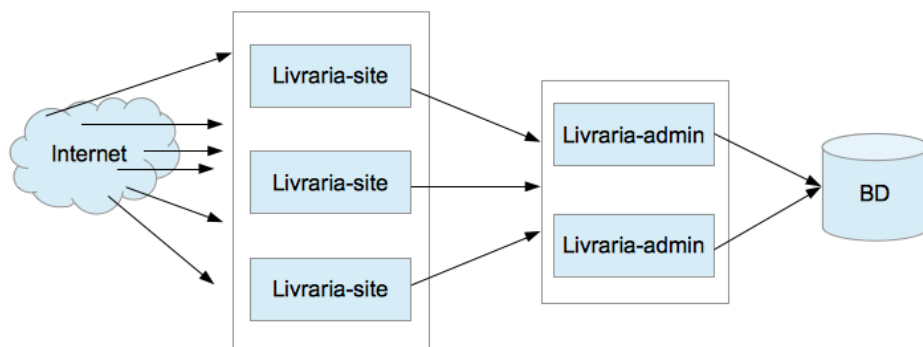
De um modo geral, prefira controllers com menos código, focados em controlar a requisição, e mantenha lógicas de negócio em componentes especializados, ou nos próprios modelos (como no `Livro` ou na `Estante`).

Capítulo 7

Integração entre Sistemas usando o VRaptor

Completamos o nosso cadastro de livros, com todas as validações necessárias para conseguirmos vendê-los sem problemas. Essa parte do sistema será instalada numa rede protegida, com acesso restrito às pessoas que podem cadastrar os livros.

A segunda parte do nosso sistema é a parte que vai vender os livros cadastrados, o site da livraria. Essa parte do sistema será instalada em outro servidor, e cuidará apenas da apresentação dos livros para a compra. Para isso, ela precisará dos dados dos livros cadastrados pelo admin do sistema.

Existem diversas formas de integrar esses dois sistemas, desde jogar arquivos numa pasta da rede até usar Web Services, e essa integração depende muito de como os sistemas foram implementados. No nosso caso, como os dois sistemas usam VRaptor, existe uma forma mais natural. Antes de falar sobre ela, vamos ver com mais detalhes como será essa integração.

Nossos dois sistemas são o **livraria-admin**, que cuidará do cadastro dos livros, e o **livraria-site**, que mostrará os livros num site para vendê-los. O **livraria-site** não terá acesso ao banco de dados, logo, para conseguir os dados dos livros terá que consultar o **livraria-admin**. Com isso, conseguimos evoluir as duas partes do sistema independentemente, tanto no código quanto no deploy. Ao atualizar o **livraria-admin** não precisamos reiniciar o **livraria-site** e vice-versa.

Além disso, os dois sistemas têm requisitos diferentes. O **livraria-site** será acessado por milhares de pessoas ao mesmo tempo, enquanto o **livraria-admin** será acessado apenas por algumas pessoas que têm acesso ao cadastro. O **livraria-admin** precisa de acesso ao banco de dados, de um ambiente transacional, de controle de acesso, enquanto o **livraria-site** ficará aberto ao público geral, servindo páginas que mostram os livros e guardando os carrinhos de compras dos usuários. São responsabilidades bem diferentes que deveriam ser feitas por sistemas diferentes.

O grande problema agora é: como passar os dados dos livros de um sistema para o outro? O VRaptor é um framework Web, que nos possibilita executar operações dentro do sistema a partir de URLs no nosso navegador. Por exemplo, ao acessar http://localhost:8080/livraria-admin/livros/lista caímos no método `lista` do `LivrosController`. Ao final do método, é mostrada uma página que foi gerada a partir do `/WEB-INF/jsp/livros/lista.jsp`. O ponto de partida é uma requisição a uma URL que começa com **http://**, ou seja, estamos usando um protocolo chamado HTTP.

Em geral, usamos bastante esse protocolo quando estamos navegando na internet, junto com a sua versão mais segura, o HTTPS, para acessarmos páginas dos sites que visitamos. No entanto, podemos usar o mesmo HTTP para acessar outros da-

dos, além de páginas Web, como imagens, arquivos para download, músicas, enfim, qualquer dado relevante para nós.

Usando o protocolo HTTP podemos disponibilizar os dados dos livros no **livraria-admin**. Para isso, precisamos criar uma URL que executará um código e nos dará esses dados que precisamos. O jeito de fazer isso no VRaptor é criar um método num controller:

```
@Resource
public class IntegracaoController {

  public void listaLivros() {
    // ...
  }
}
```

Temos a URL `/integracao/listaLivros` que executará o método e, por padrão, redirecionará para o jsp correspondente. Geralmente usamos jsps para criar páginas HTML, mas nesse caso queremos retornar apenas os dados dos livros. O que significa retornar os dados dos livros? Esses dados precisam ser passados de alguma forma para o outro sistema.

No mundo ideal, poderíamos passar uma `List<Livro>` de um sistema para o outro. Porém, essa lista é um objeto que reside na memória do servidor do **livraria-admin**, e não é possível passar exatamente esse objeto para o **livraria-site**, que é outra aplicação, está em outro servidor, em outra JVM.

7.1 Serializando os objetos

Se não podemos passar o mesmo objeto de um lado para o outro, o melhor que podemos fazer é extrair os dados do objeto em um formato qualquer e ler esses dados no outro sistema, transformando-os de volta em objetos. Chamamos esse processo, respectivamente, de serialização e deserialização do objeto em questão. Para fazer isso, precisamos definir um formato que as duas aplicações entendam e consigam escrever e ler o objeto.

O formato pode ser qualquer coisa, por exemplo, um arquivo texto formatado posicionalmente, um arquivo binário num formato que você mesmo inventou ou o objeto serializado com o próprio Java (com `Serializable`s e `Object` streams). Um formato bom, no entanto, é um formato que seja suportado facilmente nos dois sistemas, de preferência sem que seja preciso escrever o *parser* desse formato. Melhor

ainda se puder ser consumido em qualquer linguagem de programação, assim ficaríamos livres para escrever o site como bem entendermos, como em *Ruby on Rails*, *Play* no *Scala* ou *ASP*.

Um dos formatos que atende isso bem é o bom e velho XML. Conseguimos ler e gerar esse formato em qualquer linguagem de programação que seja usável em projetos de verdade. Um livro representado em XML seria algo parecido com:

```
<livro>
  <isbn>85-336-0292-8</isbn>
  <titulo>O Senhor dos Anéis</titulo>
  <autor>J. R. R. Tolkien</autor>
  <preco>130.00</preco>
</livro>
```

Por ser um formato tão conhecido e usado, o VRaptor possui uma forma fácil de serializar qualquer objeto em XML. O que queremos é mudar o resultado padrão e gerar o XML, então usamos o `Result`:

```
@Resource
public IntegracaoController {

  private Estante estante;
  private Result result;

  public IntegracaoController(Estante estante, Result result) {
    this.estante = estante;
    this.result = result;
  }

  public void listaLivros() {
    List<Livro> livros = estante.todosOsLivros();

    result.use(Results.xml()).from(livros, "livros").serialize();
  }
}
```

Assim, ao acessarmos a URL `/integracao/listaLivros`, não veremos mais uma página HTML, e sim um XML parecido com:

```
<livros>
  <livro>
```

```
    <id>1310</id>
    <isbn>85-336-0292-8</isbn>
    <titulo>O Senhor dos Anéis</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>130.00</preco>
  </livro>
  <livro>
    <id>42</id>
    <isbn>9789728839130</isbn>
    <titulo>O Guia dos Mochileiros das Galáxias</titulo>
    <autor>Douglas Adams</autor>
    <preco>90.00</preco>
  </livro>
</livros>
```

Explicando um pouco melhor a chamada:

```
// Usar como resultado um xml
result.use(Results.xml())
  // a partir do objeto livros, com o nome "livros"
  .from(livros, "livros")
  // serializa o objeto e joga na resposta.
  .serialize();
```

Esse método `serialize()` pode parecer um pouco estranho num primeiro momento, mas existe, pois o VRaptor tem uma convenção para serializar o objeto. Por padrão, só são serializados atributos simples do objeto, como números, datas, enums e Strings. Se o objeto tiver algum atributo que seja, por exemplo, um outro objeto da aplicação ou uma lista, para serializarmos esse objeto precisamos explicitamente incluí-lo na serialização. Por exemplo, se tivermos a classe `Autor`:

```
enum Pais { BRASIL, ESTADOS_UNIDOS, REINO_UNIDO  }
class Autor {

  private String nome;
  private Calendar dataNascimento;
  private Integer numeroDeLivros;
  private Pais naturalidade;

  private List<Livro> livros;
  private Livro ultimoLivro;
```

```
}
```

Ao serializarmos um autor da forma padrão:

```
Autor autor = // busca do banco
result.use(Results.xml()).from(autor).serialize();
```

Só serão serializados os atributos simples:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1892-01-03</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
</autor>
```

Se quisermos serializar, por exemplo, o último livro, precisamos pedir para incluí-lo:

```
result.use(Results.xml()).from(autor).include("ultimoLivro").serialize();
```

Assim, a serialização ficaria:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1892-01-03</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
  <ultimoLivro>
    <id>233</id>
    <isbn>85-336-1165-X</isbn>
    <titulo>Silmarillion</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>13.00</preco>
  </ultimoLivro>
</autor>
```

A regra de somente serializar atributos simples vale para o objeto incluído também, ou seja, o último livro só terá os atributos simples. Se quisermos incluir um atributo específico do livro, fazemos `.include("ultimoLivro", "ultimoLivro.umAtributo")` Se quisermos, também, serializar a lista de livros desse autor, incluímos o campo respectivo:

```
result.use(Results.xml()).from(autor)
      .include("ultimoLivro", "livros").serialize();
```

A serialização:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1876-04-05</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
  <livros>
    <livro>
      <id>233</id>
      <isbn>85-336-1165-X</isbn>
      <titulo>Silmarillion</titulo>
      <autor>J. R. R. Tolkien</autor>
      <preco>13.00</preco>
    </livro>
    <livro>
      <id>1310</id>
      <isbn>8533613377</isbn>
      <titulo>O Senhor dos Anéis - A sociedade do anel</titulo>
      <autor>J. R. R. Tolkien</autor>
      <preco>60.00</preco>
    </livro>
    <!-- outros livros -->
  </livros>
  <ultimoLivro>
    <id>233</id>
    <isbn>85-336-1165-X</isbn>
    <titulo>Silmarillion</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>13.00</preco>
  </ultimoLivro>
</autor>
```

Uma outra possibilidade é pedir para serializar a árvore inteira de objetos, ou seja, todos os atributos, recursivamente, com o método `recursive`:

```
result.use(Results.xml()).from(autor).recursive().serialize();
```

Nada será excluído da serialização. Muito cuidado, pois isso pode gerar respostas gigantes e pesadas:

```
<autor>
  ...
  <ultimoLivro>
    ...
    <editora>
      ...
      <colecoes>
        <colecao>
          ...
          <lojas>
            ...
             ...
          </lojas>
        </colecao>
      </colecoes>
    </editora>
  </ultimoLivro>
</autor>
```

Ou pior, se existisse um ciclo nessa árvore de objetos, por exemplo, se o livro tivesse uma referência para o autor, a recursão seria infinita:

```
<autor>
  <name>J. R. R. Tolkien</name>
  ...
  <ultimoLivro>
    <name>Senhor dos Anéis</name>
    ...
    <autor>
      <name>J. R. R. Tolkien</name>
      ...
      <ultimoLivro>
        <name>Senhor dos Anéis</name>
        ...
        <autor>
          <name>J. R. R. Tolkien</name>
          ...
          <ultimoLivro>
            <name>Senhor dos Anéis</name>
            ...
```

Para evitar isso, acontecerá um erro caso exista um ciclo na árvore de objetos.

Em todo caso, pense muito bem antes de usar o `.recursive()`, pois a resposta pode ficar maior que o necessário. Caso a configuração do XML fique muito complicada, considere criar uma classe que expõe somente os dados que você quer serializar — um DTO (Data Transfer Object, do livro PoEAA do Martin Fowler, ou http://martinfowler.com/eaaCatalog/dataTransferObject.html) .

Voltando ao resultado anterior, sem o `recursive()`, cada livro possui o nome do autor, que é um pouco redundante, já que estamos serializando o livro. Se quisermos remover o atributo `autor` da serialização, podemos usar o método `.exclude`:

```
result.use(Results.xml())
      .include("ultimoLivro")
      .exclude("ultimoLivro.autor")
      .serialize();
```

Assim, a resposta ficaria:

```
<autor>
  <nome>J. R. R. Tolkien</nome>
  <dataNascimento>1876-04-05</dataNascimento>
  <naturalidade>REINO_UNIDO</naturalidade>
  <ultimoLivro>
    <id>233</id>
    <isbn>85-336-1165-X</isbn>
    <titulo>Silmarillion</titulo>
    <preco>13.00</preco>
  </ultimoLivro>
</autor>
```

7.2 Recebendo os dados no sistema cliente

Uma das vantagens de expor um serviço da maneira acima é que o cliente desse serviço pode ser qualquer tipo de sistema, escrito em qualquer linguagem de programação com a qual é possível executar uma requisição HTTP e consumir XML. E como HTTP é um dos protocolos mais usados e XML é um dos formatos mais adotados, praticamente todas as linguagens de programação saberão trabalhar com eles.

No nosso caso, o sistema cliente será uma aplicação Java também escrita usando o VRaptor: o **livraria-site**. Antes de começar a integração, vamos escrever a página inicial do site, que mostrará os livros numa vitrine. No projeto **livraria-site**, vamos criar o `HomeController`:

```
@Resource
public class HomeController {

  public void inicio() {
  }

}
```

Precisamos mostrar na página dessa lógica uma lista de livros, que será consumida do serviço do **livraria-admin**. Devemos colocar o código que consome o serviço dentro do método `inicio`? Precisaremos também de dados de livros na página que mostra o livro, na página do carrinho de compras, na página de finalização da compra e em várias outras, então não vale a pena repetir esse código por todo canto.

Para evitar essa repetição, vamos criar um componente do sistema responsável por acessar os dados dos livros, como fizemos no **livraria-admin**, um **repositório** de livros. A diferença é que, agora, o acesso aos dados se dá pelo serviço disponibilizado pelo **livraria-admin**, e não pelo banco de dados. Podemos usar aqui, também, a `Estante`, nosso repositório de livros. Mas para não causar confusões, vamos usar outra abstração para um conjunto de livros: um **Acervo**, que nos dará acesso aos livros do sistema, que estão cadastrados no **livraria-admin**.

Vamos criar a interface `Acervo` no projeto **livraria-site**.

```
public interface Acervo {

  List<Livro> todosOsLivros();

}
```

Precisamos também ter o `Livro` do lado do site, já que trabalhamos com objetos, e não com XMLs, no resto do sistema. A classe `Livro` não precisa ser necessariamente idêntica nos dois sistemas, já que alguns dados podem não fazer sentido para o site, por exemplo, se guardássemos preço de custo, margem de lucro, quantidade em estoque ou outras coisas. No nosso caso, podemos copiar a classe `Livro` do admin para o site, já que vamos usar todos os dados do livro. Podemos, no entanto, remover as anotações da JPA, já que não vamos salvá-lo no banco de dados.

Podemos agora implementar a *home* do site, disponibilizando a lista de livros para a jsp:

```
@Resource
public class HomeController {

  private Acervo acervo;
  private Result result;

  public HomeController(Acervo acervo, Result result) {
    this.acervo = acervo;
    this.result = result;
  }

  public void inicio() {
    this.result.include("livros", acervo.todosOsLivros());
  }

}
```

E no JSP padrão (`WEB-INF/jsp/home/inicio.jsp` dentro de `livraria-site/src/main/webapp`) acrescentar em alguma parte a lista dos livros:

```
<h3>Veja as últimas ofertas para você:</h3>

<ul class="livros">
  <c:forEach items="${livros}" var="livro">
    <li class="livro">${livro.titulo} - R$ ${livro.preco}</li>
  </c:forEach>
</ul>
```

7.3 Consumindo os dados do admin

Com a página inicial implementada, precisamos agora alimentá-la com os dados dos livros, que virão do **livraria-admin**. Precisamos implementar o `Acervo` consumindo os serviços desse sistema. Vamos criar a classe `AcervoNoAdmin`, como um componente que implementa a `Acervo`:

```
package br.com.casadocodigo.livraria.site.servico;

@Component
public class AcervoNoAdmin implements Acervo {
```

```
  @Override
  public List<Livro> todosOsLivros() {
    //...
  }
}
```

A **livraria-admin** possui um serviço que criamos para retornar a lista de todos os livros. Esse serviço foi criado usando o protocolo HTTP, que é o que usamos na Web. Precisamos de algo que consiga consumir um serviço HTTP, na URL que nos retorna o XML dos livros.

Se pensarmos nos serviços Web tradicionais, teríamos que criar um *endpoint* do serviço, que ia gerar um documento com todas as operações possíveis e, na aplicação cliente, teríamos que criar várias classes para conseguir consumir esse serviço.

No nosso caso, estamos usando o próprio VRaptor pra expor esse serviço, como se fôssemos usá-lo no navegador, usando o protocolo HTTP. Para isso, tudo o que precisamos é de um cliente HTTP, que existe em qualquer linguagem de programação que formos usar. Em Java, existe um jeito fácil de fazer uma requisição HTTP e receber o resultado, usando a classe `java.net.URL`:

```
URL url = new URL("http://localhost:8080/livraria-admin" +
  "/integracao/listaLivros");
InputStream resposta = url.openStream();
```

Ou seja, passando a URL do serviço que nos retorna a lista de livros, conseguimos executar uma requisição a essa URL, recebendo a resposta como um `InputStream`. No entanto, muitas coisas podem dar errado no meio do caminho. Por exemplo, a URL pode estar incorreta, pode estar apontando para um lugar que não existe, o servidor pode estar fora do ar, a rede pode ter caído etc. Por esse motivo, o código acima lança exceções que precisam ser tratadas.

```
try {
  URL url = new URL("http://localhost:8080/livraria-admin" +
    "/integracao/listaLivros");
  InputStream resposta = url.openStream();
} catch (MalformedURLException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Esse é um código de muito baixo nível para estar no Acervo, que é uma abstração de alto nível, então vamos criar um componente para nos ajudar a fazer requisições HTTP, retornando-nos uma String com a resposta. Esse componente será o ClienteHTTP, com a implementação usando URL.

```
package br.com.casadocodigo.livraria.site.servico;

public interface ClienteHTTP {
  String get(String url);
}

@Component
public class URLClienteHTTP implements ClienteHTTP {

  @Override
  public String get(String url) {
    try {
      URL servico = new URL(url);
      InputStream resposta = servico.openStream();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Precisamos agora converter essa resposta de InputStream para String. Existem vários jeitos de fazer isso, em várias bibliotecas diferentes. Vamos usar a classe com.google.common.io.CharStreams, da biblioteca **Google Guava** que já é dependência do VRaptor e contém vários utilitários que melhoram vários aspectos do Java. Essa classe consegue transformar um Reader numa String, e para transformar o InputStream num Reader usamos o InputStreamReader do próprio java.

```
Reader reader = new InputStreamReader(resposta);
String respostaEmString = CharStreams.toString(reader);
```

Integrando com o resto do código, ficaria:

```
public String get(String url) {
  try {
```

```
    URL servico = new URL(url);
    InputStream resposta = servico.openStream();
    Reader reader = new InputStreamReader(resposta);
    return CharStreams.toString(reader);
  } catch (MalformedURLException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Esse código ainda não compila, pois precisamos decidir o que fazer caso ocorram exceptions. Temos dois tipos de `Exception` sendo lançadas aqui: `MalformedURLException` — quando a url não é válida, e `IOException` — quando aconteceu algum erro de comunicação com o servidor durante a requisição.

A `MalformedURLException`, no nosso caso, seria culpa do programador, já que as urls vão vir do código, logo podemos supor que elas só vão acontecer em tempo de desenvolvimento e simplesmente a lançarmos dentro de alguma `RuntimeException`.

```
} catch (MalformedURLException e) {
  throw new IllegalArgumentException("A url " + url +
        " está inválida, corrija-a!", e);
}
```

Já a `IOException` foge do nosso controle e, se faz parte do nosso processo se recuperar de alguma forma caso o servidor esteja fora do ar, precisamos deixar isso no contrato do nosso cliente http, ou seja, da interface `ClienteHTTP`. Lançar `IOException` talvez seja genérico demais — podemos optar por criar uma exceção mais específica, como `ServidorIndisponivelException`:

```
public class ServidorIndisponivelException extends RuntimeException {
  public ServidorIndisponivelException(String url, Exception e) {
    super("Erro ao fazer requisição ao servidor na url " + url, e);
  }
}

public interface ClienteHTTP {
  public String get(String url) throws ServidorIndisponivelException;
}
```

E na hora de fazer o `catch`, usamos a nossa exceção:

```
} catch (IOException e) {
  throw new ServidorIndisponivelException(url, e);
}
```

A `ServidorIndisponivelException` está como `RuntimeException` para não obrigarmos o usuário do `ClienteHTTP` a tratá-la. Mas deixamos a exceção declarada na interface, para que o usuário saiba que tem a opção de se recuperar desse caso específico. Se quisermos obrigar a sempre tratar esse caso, trocamos o `extends` para `Exception`, mas cuidado que isso pode gerar muitos métodos que, ou relançam essa exceção declarando o `throws ServidorIndisponivelException`, ou fazem um `try..catch` relançando a exceção dentro de uma `RuntimeException`, que tornam o código bem chato de manter e evoluir.

7.4 Transformando o XML em objetos

Com o nosso `ClienteHTTP` implementado, agora nosso código da `AcervoNoAdmin` está assim:

```
@Component
public class AcervoNoAdmin implements Acervo {

  private ClienteHTTP http;

  public AcervoNoAdmin(ClienteHTTP http) {
    this.http = http;
  }

  @Override
  public List<Livro> todosOsLivros() {
    String url = "http://localhost:8080/livraria-admin" +
        "/integracao/listaLivros";
    String resposta = http.get(url);

    return null;
  }

}
```

A resposta será um XML parecido com:

```
<livros>
  <livro>
    <id>1310</id>
    <isbn>8533613377</isbn>
    <titulo>O Senhor dos Anéis</titulo>
    <autor>J. R. R. Tolkien</autor>
    <preco>130.00</preco>
  </livro>
  <livro>
    <id>42</id>
    <isbn>9789728839130</isbn>
    <titulo>O Guia dos Mochileiros das Galáxias</titulo>
    <autor>Douglas Adams</autor>
    <preco>90.00</preco>
  </livro>
</livros>
```

Precisamos agora ler essa resposta, que contém um XML com a lista de livros, e transformá-la em objetos `List<Livro>`, para podermos usar esses dados no restante do sistema. Existem várias bibliotecas que fazem isso para nós, uma delas é o **XStream**, que já vem como dependência do VRaptor e é a biblioteca que ele usa para gerar e consumir XMLs (e JSONs). A ideia do XStream é que ele seja bem simples de usar, com o mínimo de configurações.

```
XStream xstream = new XStream();
Object object = xstream.fromXML(resposta);
```

A convenção do XStream para conseguir deserializar a resposta em um objeto é que o nó raiz tenha o nome da classe, ou seja, se quisermos deserializar um `Livro`, o nó raiz do XML deveria ser `<br.com.casadocodigo.livraria.modelo.Livro>`. A partir daí, sempre que possível, ele tenta usar os atributos dos objetos como nomes das tags.

Para conseguir consumir o XML que o VRaptor gerou, precisamos configurar isso no XStream, falando que o nó raiz `<livros>` é uma lista e que o nó de cada elemento da lista, `<livro>`, é um `Livro`. Fazemos isso com o método `alias`:

```
XStream xstream = new XStream();
xstream.alias("livros", List.class);
xstream.alias("livro", Livro.class);
```

Ao pedirmos para o XStream consumir o XML, temos como resposta uma `List<Livro>`:

```
List<Livro> livros = (List<Livro>) xstream.fromXML(xml);
```

Nosso método completo ficaria:

```
@Override
public List<Livro> todosOsLivros() {
  String xml = http.get("http://localhost:8080/livraria-admin" +
      "/integracao/listaLivros");

  XStream xstream = new XStream();
  xstream.alias("livros", List.class);
  xstream.alias("livro", Livro.class);

  List<Livro> livros = (List<Livro>) xstream.fromXML(xml);
  return livros;
}
```

Com isso, se conhecermos a URL da lógica do admin que retorna a lista de livros, conseguimos os dados dos livros novamente em objetos java e, então, podemos trabalhar com eles normalmente no resto do sistema.

7.5 Aproveitando melhor o protocolo HTTP - REST

Quando pensamos em serviços web tradicionais, os *Web Services SOAP*, para cada operação que o servidor suportar, é necessário criar pelo menos 4 classes no cliente, sem contar as classes de modelo intermediárias que serão trafegadas de um sistema para o outro. Isso porque precisamos ensinar o cliente a fazer cada uma das operações, qual é o formato de entrada, qual é o formato de saída, entre outras coisas. Tudo isso é definido dentro de um documento chamado de WSDL (sigla para Web Services Description Language).

Já no nosso caso, estamos usando o protocolo HTTP para implementar o serviço, e não precisamos criar classes para ensinar o sistema cliente a executar uma operação no sistema servidor. Tudo o que precisamos foi de um cliente HTTP. Isso acontece porque o HTTP já define um conjunto pequeno de operações possíveis dentro desse protocolo.

Com isso, basta que o cliente saiba executar essas operações e o servidor saiba entender essas operações fixas e, assim, nenhuma classe de infraestrutura é necessária para que um sistema se comunique com o outro. Apesar de esse conjunto de operações ser pequeno, são operações universais, que são suficientes para implementarmos qualquer tipo de serviço. Essas operações, também chamadas de método ou verbo, são: *GET*, *POST*, *PUT*, *DELETE*, *HEAD*, *OPTIONS*, *TRACE* e, recentemente, *PATCH*.

Mas como representar a operação de listar todos os livros do sistema, sendo que eu só tenho esses 8 métodos HTTP disponíveis? A ideia do HTTP é que esses métodos devem ser obrigatoriamente executados em um **recurso**, que é qualquer entidade, dado ou personagem do sistema. Esse recurso é representado por uma URI (*Unified Resource Identifier*, ou identificador único de um recurso). Uma URL é um tipo de URI, que também representa qual é o local onde podemos encontrar o recurso (o **L** é de *Location*, ou seja, local).

Cada método tem uma semântica muito bem definida possibilitando que, em conjunção com um recurso, consigamos representar qualquer operação possível do nosso sistema. Usamos dois desses métodos constantemente enquanto navegamos na web: o **GET**, ao clicarmos em links ou digitarmos endereços diretamente no navegador, e o **POST**, ao submetermos formulários. A ideia é que, se usarmos corretamente as semânticas dos métodos, podemos aproveitar toda a infraestrutura da internet, como proxies, load balancers e outros intermediários entre o cliente e o servidor.

A semântica dos métodos HTTP são:

- **GET** — Retorna as informações de um recurso. Por exemplo, `GET /livros` pode devolver todos os livros do sistema, e `GET /livro/1234`, as informações do livro de id igual a 1234. Esse método NÃO PODE alterar o estado do recurso em questão e é idempotente, ou seja, pode ser executado quantas vezes for necessário sem que isso afete a resposta. Usando esse fato, os proxies podem fazer o *cache* da resposta, evitando a sobrecarga do servidor. Robôs de busca podem indexar todo o conteúdo de um site seguindo todos os links que encontram numa página, já que eles executam o método GET na URL do link, e não alteram nada no site em questão.

- **POST** — Acrescenta informações em um recurso. Por exemplo, ao fazermos um `POST /livros`, passando os dados de um livro, estamos criando um livro novo no sistema. Esse método modifica o estado recurso em questão,

podendo criar novos recursos e não é idempotente, ou seja, se fizermos dois POSTs em `/livros`, mesmo que estejamos passando os mesmos dados, criaremos dois livros.

- **PUT** — Substitui as informações de um recurso. Se fizermos `PUT /livro/1234`, passando dados desse livro, atualizamos os seus atributos. Esse método modifica o recurso indicado, substituindo TODAS as suas informações pelas passadas no corpo da requisição. Se o recurso não existir, ele poderá ser criado.

- **PATCH** — Tem mesma semântica do `PUT`, exceto que podemos passar apenas a parte das informações que desejamos alterar no recurso, que deve existir. Esse método é mais recente e ainda não é suportado em todos os servidores.

- **DELETE** — Remove o recurso. Por exemplo, `DELETE /livro/321` remove o livro de id 321. Também é uma operação idempotente, ou seja, remover duas vezes o recurso deve ter o mesmo efeito de fazer isso uma única vez.

- **HEAD** — Parecido com o GET, mas retorna apenas os headers da resposta.

- **OPTIONS** — Retorna as operações possíveis e, possivelmente, algumas metainformações do recurso.

- **TRACE** — para fazer Debugging.

Pensando na Web "humana", dificilmente usamos todos esses métodos, principalmente porque, pela especificação do HTML, os navegadores só suportam os métodos **GET** e **POST**. Mas mesmo usando esses dois métodos é bastante importante usarmos as suas semânticas: GET para operações que não modificam recursos, e POST para as que modificam, assim os intermediários podem trabalhar da maneira correta.

O protocolo HTTP vai muito além de apenas métodos e recursos e, se estamos implementando um serviço que usa as características do HTTP dizemos que é um serviço REST (ou RESTful web service). Isso vai desde usar métodos e recursos da maneira correta até fazer negociação de conteúdo e usar hipermídia. É um assunto bastante extenso, que não será abordado em detalhes nesse livro, mas podemos dar os primeiros passos nessa direção com a ajuda do VRaptor.

7.6 Usando métodos e recursos da maneira correta

Quando criamos um controller no VRaptor, existe uma convenção que gera a URL de cada método desse controller. Por exemplo, no nosso `LivrosController`:

```
@Resource
public class LivrosController {
   public void lista() {...}
   public void salva(Livro livro) {...}
   public void edita(String isbn) {...}
}
```

As URLs dos métodos `lista`, `salva` e edita terminam em `/livros/lista`, `/livros/salva` e `/livros/edita`, respectivamente. É bastante usual darmos os nomes dos métodos de um objeto usando verbos, afinal estamos executando ações. Mas, pensando em REST, URLs (ou URIs) deveriam denotar recursos, que são substantivos, e não verbos. Mas se estamos usando substantivos, onde ficam as ações? Nos métodos (ou verbos) HTTP.

No método `lista`, estamos retornando as informações de todos os livros, então o recurso pode ser `/livros`. A lista não altera nada no sistema, portanto o método HTTP é o **GET**. Para alterarmos a URL de um método do controller, podemos usar a anotação `@Path` do VRaptor:

```
@Path("/livros")
public void lista() {...}
```

A partir do momento em que fazemos isso, a URL `/livros/lista` já não corresponde mais a esse método, que deverá ser acessado por `/livros` somente. Por padrão, o VRaptor aceita qualquer verbo HTTP nos métodos (mesmo nos que não tem o `@Path`), logo, se quisermos indicar que esse método só pode ser acessado via `GET`, usamos a anotação `@Get`:

```
@Get @Path("/livros")
public void lista() {...}
```

As anotações de métodos HTTP também recebem a URI como parâmetro. Podemos fazer simplesmente:

```
@Get("/livros")
public void lista() {...}
```

Já o método salva acrescenta um Livro novo ao sistema, logo não podemos usar o GET. O recurso ainda é a lista dos livros do sistema, então podemos usar POST /livros para acessar esse método. Assim como temos o @Get para requisições GET, usamos o @Post para requisições POST:

```
@Post("/livros")
public void salva(Livro livro) {...}
```

Repare que tanto o método lista quanto o método salva usam a mesma URI, mas selecionamos qual método será invocado dependendo do método HTTP da requisição. No método edita não estamos mais tratando de todos os livros do sistema, mas sim de um livro específico. Pra indicar isso numa URI, precisamos que ela contenha uma informação que identifique unicamente o livro, no nosso caso, o isbn. Esse método mostra um formulário com os dados do Livro para a edição, então não estamos alterando nada no sistema, ou seja, podemos usar o método GET.

Como podemos editar qualquer um dos livros do sistema, precisamos de uma URI para cada um deles. Não é viável declararmos todas as URIs possíveis no método do controller. Precisamos declarar um *template* de URI que vai cair no método edita, ou seja, uma URI que contém uma variável no meio, que será o ISBN do livro. Para declarar uma variável no meio de uma URI no VRaptor usamos {} com o nome da variável capturada dentro. No caso do método edita ficaria:

```
@Get("/livros/{isbn}")
public void edita(String isbn) {...}
```

Desse modo, ao acessarmos a URI /livros/12345, o VRaptor vai invocar o método edita, capturando o valor 12345 na variável isbn. Como o método recebe um parâmetro com esse nome, o valor 12345 será passado para o método. Como o isbn faz parte da URI, não é possível invocar esse método via HTTP sem passar um isbn, assim não precisamos verificar se esse parâmetro foi passado mesmo na requisição. Nosso método edita está assim:

```
@Get("/livros/{isbn}")
public void edita(String isbn) {
  Livro livroEncontrado = estante.buscaPorIsbn(isbn);
  if (livroEncontrado == null) {
    result.notFound();
  } else {
    result.include(livroEncontrado);
```

```
    result.of(this).formulario();
  }
}
```

O que acontece se passarmos um `isbn` que não existe, por exemplo `GET /livros/bazinga`? O mundo inteiro já conhece uma resposta apropriada para um endereço que não existe na Web: a famosa página 404. Esse número nada mais é do que um dos status codes possíveis do protocolo HTTP, e significa que o recurso não foi encontrado. É exatamente o nosso caso, por isso que simplesmente podemos redirecionar para a página 404 do sistema caso um livro com o ISBN passado não for encontrado, usando o método `notFound` do `Result`.

Se tivermos um método `altera` no controller, podemos também receber o identificador do objeto na URI, já populando o objeto, usando a mesma convenção de nomes que a dos parametros de formulário. Nesse caso, como estamos alterando os dados do `Livro`, podemos usar o método `PUT`

```
@Put("/livros/{livro.isbn}")
public void altera(Livro livro) {...}
```

Assim o `livro` virá com o `isbn` populado com o valor passado na URI, e o resto dos atributos populados usando o corpo da requisição. Da mesma forma, se quisermos remover o livro, podemos usar o método `DELETE`:

```
@Delete("/livros/{livro.isbn}")
public void remove(Livro livro) {...}
```

Passando a pensar nas URIs como recursos fica muito mais simples para um cliente do sistema consumir os serviços Web. Não é mais necessário conhecer a URI de todos os métodos do serviço, somente conhecer as URIs correspondentes aos recursos é o suficiente. Sabendo que temos o recurso `/livros`, o cliente sabe que `GET /livros` significa obter os dados dos livros e `POST /livros` significa criar um livro novo. Se o recurso é um livro específico, `/livros/1234`, o cliente sabe que para obter os dados do livro deve usar `GET`, para editar os dados deve usar `PUT` (ou `PATCH` se for editar apenas alguns campos) e para remover o livro deve usar `%DELETE`.

Alterando o link para a edição de livros

Na listagem de livros estamos usando o seguinte link:

```
<a href="${linkTo[LivrosController].edita }?isbn=${livro.isbn}">
  Modificar
</a>
```

Usando o recurso de uma maneira REST, estamos passando o ISBN diretamente na url, e não como parâmetro. Nesse caso, podemos passar parâmetros para o método do controller que serão usados na formação da url. Fazemos isso usando colchetes, ao invés de parênteses:

```
<a href="${linkTo[LivrosController].edita[livro.isbn]}">
  Modificar
</a>
```

Essa passagem de parâmetros só será efetiva se ele for usado diretamente na URL.

Resolvendo conflitos de rotas

Ao criar a rota do método edita, usamos `@Get("/livros/{isbn}")`. O problema é que a rota para o método formulário é, implicitamente, `@Path("/livros/formulario")`, que também casa com o padrão `@Get("/livros/{isbn}")`, com o ISBN valendo `formulario`. Isso, a princípio, vai gerar um 404, pois se o método formulário não tem nada anotado, a sua rota tem menos prioridade. Mas caso ele estivesse anotado com `@Get("/livros/formulario")`, receberíamos um erro:

```
There are two rules that matches the uri '/livros/formulario'
  with method GET:
[[FixedMethodStrategy: /livros/formulario LivrosController.formulario()
  [GET]],
[FixedMethodStrategy: /livros/{isbn} LivrosController.edita(String)
  [GET]]]
with same priority. Consider using @Path priority attribute.
```

Ou seja, ele fala que o método `formulario` e o método `edita` conseguem tratar a uri `/livros/formulario`, com a mesma prioridade. Na última frase,

ele dá a dica de usar o atributo `priority` da anotação `@Path` para resolver essa ambiguidade. Para isso, devemos mudar o método edita para:

```
@Get @Path(value="/livros/{isbn}", priority=Path.LOWEST)
public void edita(String isbn) {...}
```

Assim, essa rota tem a menor prioridade, e a `/livros/formulario` será usada para o método `formulario` do controller. Caso o ISBN fosse um número, essa configuração não seria necessária:

```
@Get("/livros/{isbn}")
public void edita(Long isbn) {...}
```

Nesse caso, como o VRaptor sabe que o ISBN é um número (Long), ele sabe que a última parte do caminho tem que conter apenas dígitos. Dessa forma, `/livros/1234` cai no método edita, mas `/livros/abcd` dá 404 direto.

7.7 Usando REST no navegador

Pensando em REST estamos facilitando a vida do cliente que vai consumir os serviços da nossa aplicação. Mas nem sempre esses clientes são outras aplicações. Quando uma pessoa acessa o sistema usando um navegador, ela é o cliente do sistema, usando o navegador como cliente HTTP. Por isso, podemos aplicar as ideias do REST à Web humana também, para implementarmos as interações com o usuário usando HTML e javascript.

Uma limitação forte dos navegadores é que só conseguimos fazer requisições GET e POST nativamente. Isso significa que, se quisermos fazer um formulário para alterar um livro, não conseguimos fazer:

```
<form action="/livros/1234" method="PUT">
```

Mas para simular o método PUT, o VRaptor (e alguns outros frameworks Web) suporta receber um parâmetro a mais na requisição indicando qual é o método real que queremos executar. No VRaptor, devemos deixar o formulário como POST e passar um parâmetro chamado `_method` com valor `PUT`:

```
<form action="/livros/1234" method="post">
  <input type="hidden" name="_method" value="PUT"/>
```

Capítulo 8

Download e Upload de arquivos

8.1 Enviando arquivos para o servidor: Upload

Quando implementamos o cadastro de livros, ficou faltando uma parte importante para podermos mostrar os livros na nossa vitrine virtual: a capa dos livros. Essa capa será uma imagem cujo upload vamos fazer para o servidor, para depois podermos baixá-la.

Para isso, precisamos colocar um campo no formulário do livro para incluirmos a capa. Como esse formulário vai conter um campo para upload, precisamos mudar o enctype para `multipart/form-data`. Além disso, criaremos o campo de upload como uma tag `<input>` com `type="file"`:

```
<form action="${linkTo[LivrosController].salva}"
    method="post" enctype="multipart/form-data">
  ...
  Capa: <input type="file" name="capa" />
  ...
</form>
```

Precisamos acrescentar a biblioteca `commons-fileupload` e a `commons-io` para que o VRaptor consiga tratar uploads corretamente. Se estivermos usando um servidor compatível com Servlet 3, essas bibliotecas podem não ser necessárias. No nosso caso, em que estamos usando o ivy para gerenciar dependências, precisamos acrescentar as seguintes linhas no ivy.xml e rodar o `ant resolve`:

```
<dependency org="commons-fileupload" name="commons-fileupload"
  rev="1.2.1" conf="default"/>
<dependency org="commons-io" name="commons-io"
  rev="1.3.2" conf="default"/>
```

Do lado do Controller, para recebermos a capa, precisamos acrescentar um parâmetro a mais no método `salva`, com o mesmo nome do input e com o tipo `UploadedFile`:

```
public class LivrosController {

  //...
  @Post("/livros")
  public void salva(Livro livro, UploadedFile capa) {

  }

}
```

Esse `UploadedFile` nos dá o conteúdo do arquivo e algumas informações que podemos usar para salvá-lo corretamente:

```
capa.getFile(); // um InputStream com o conteúdo do arquivo
capa.getFileName(); // o nome do arquivo
capa.getContentType(); // o tipo do arquivo. Ex: image/png
capa.getSize(); // o tamanho do arquivo.
```

Para salvar esse arquivo temos muitas opções. Podemos escolher uma pasta do servidor e salvar todas as capas nessa pasta. Podemos usar um servidor de arquivos para isso, como um FTP ou um serviço externo como o Dropbox ou o S3 da Amazon. Podemos, ainda, salvar esse arquivo direto no banco de dados. Para não se preocupar com isso do lado do Controller, vamos criar uma interface responsável por salvar arquivos, o `Diretorio`, onde poderemos salvar uma imagem e recuperá-la depois.

```
public interface Diretorio {

  URI grava(Arquivo arquivo);

  Arquivo recupera(URI chave);

}
```

O que vamos gravar é um `Arquivo` que guardará o conteúdo do arquivo junto com algumas metainformações e nos retornará uma `URI`, identificando o local onde o arquivo foi armazenado. Para recuperá-lo, basta passar essa `URI`, que recebemos de volta o `Arquivo` gravado. A classe `Arquivo` conterá o nome do arquivo, o conteúdo, o tipo e a data de modificação:

```
public class Arquivo {

  private String nome;
  private byte[] conteudo;
  private String contentType;
  private Calendar dataModificacao;

  public Arquivo(String nome, byte[] conteudo, String contentType,
      Calendar dataModificacao) {
    this.nome = nome;
    this.conteudo = conteudo;
    this.contentType = contentType;
    this.dataModificacao = dataModificacao;
  }

  //getters
}
```

Assim, podemos receber o `Diretorio` no construtor do `LivrosController` e guardar o arquivo, salvando a `URI` da imagem no `Livro`:

```
public LivrosController(Estante estante, Diretorio imagens,
    Result result, Validator validator) {
  this.estante = estante;
  this.imagens = imagens;
  this.result = result;
  this.validator = validator;
```

```
}

@Transacional
@Post("/livros")
public void salva(final Livro livro, UploadedFile capa)
    throws IOException {
  validator.validate(livro);
  validator.onErrorRedirectTo(this).formulario();

  if (capa != null) {
    URI imagemCapa = imagens.grava(new Arquivo(
      capa.getFileName(),
      ByteStreams.toByteArray(capa.getFile()),
      capa.getContentType(),
      Calendar.getInstance()));

    livro.setCapa(imagemCapa);
  }

  estante.guarda(livro);

  result.include("mensagem", "Livro salvo com sucesso!");
  result.redirectTo(this).lista();
}
```

Para conseguirmos salvar corretamente a `URI` no `Livro`, vamos adaptá-la para salvar como `String`, usando o getter e o setter:

```
public class Livro {
  //...
  private String capa;

  public URI getCapa() {
    if (capa == null) return null;
    return URI.create(capa);
  }

  public void setCapa(URI capa) {
    this.capa = capa == null ? null : capa.toString();
  }
}
```

Para salvar de fato a imagem da capa, precisamos de uma implementação real do `Diretorio`. Para que não seja necessário criar uma infraestrutura adicional, vamos usar o banco de dados para salvar os dados da imagem. Criaremos o `DiretorioNoBD` que, para funcionar, é preciso adaptar a classe `Arquivo` para virar uma entidade:

```
@Entity
public class Arquivo {

  @Id @GeneratedValue
  private Long id;

  @Lob
  private byte[] conteudo;

  // para a JPA não reclamar
  Arquivo() {}

  // resto dos campos e getters
}

@Component
public class DiretorioNoBD implements Diretorio {

  private EntityManager manager;

  public DiretorioNoBD(EntityManager manager) {
    this.manager = manager;
  }

  @Override
  public URI grava(Arquivo arquivo) {
    return null;
  }

  @Override
  public Arquivo recupera(URI chave) {
    return null;
  }

}
```

Para implementar o método `grava`, vamos salvar o arquivo no banco de dados. A `URI` que vamos adotar como resposta será do tipo `bd://<id da imagem no bd>`, afinal, estamos gravando o arquivo no banco de dados. Assim, se o arquivo tiver o id 1234, a `URI` será `bd://1234`. O método `grava` ficaria:

```
@Override
public URI grava(Arquivo arquivo) {
  manager.persist(arquivo);

  return URI.create("bd://" + arquivo.getId());
}
```

Com isso, podemos acessar o formulário de livros e cadastrar a capa para algum deles.

Bug no VRaptor 3.4.0 até 3.5.1

Se você usa o VRaptor nas versões entre 3.4.0 e 3.5.1 com o Guice como container de dependências, pode ser que você receba um erro ao fazer um upload, falando que não é possível encontrar um converter para a interface `UploadedFile`. Para resolver esse problema, atualize para uma versão maior ou igual à 3.5.2, ou crie a seguinte classe na sua aplicação:

```
@Convert(UploadedFile.class)
public class UploadedFileConverter implements
        Converter<UploadedFile> {

  private HttpServletRequest request;
  public UploadedFileConverter(HttpServletRequest request) {
    this.request = request;
  }

  @Override
  public UploadedFile convert(String value,
      Class<? extends UploadedFile> type,
      ResourceBundle bundle) {
    return type.cast(request.getAttribute(value));
  }

}
```

8.2 Recuperando os arquivos salvos: Download

Se quisermos mostrar a capa do livro na listagem, precisamos conseguir o conteúdo da imagem a partir da `URI` que salvamos. No HTML, para mostrar uma imagem usamos a tag `img`:

```
<img src="<url da imagem>" />
```

Precisamos de uma URL da aplicação que nos retorne a capa do livro e de um método de controller que vai retornar a imagem. Vamos, então, criar o método `capa` no `LivrosController`, respondendo pela `URL /livros/<isbn>/capa`:

```
@Get("/livros/{isbn}/capa")
public void capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);
  // retorna a imagem
}
```

Agora, para retornar os dados da imagem, não basta incluir a imagem no `result`. Precisamos que a resposta inteira da requisição seja a imagem. Para isso, precisamos fazer o **Download** da imagem. Para fazer um Download usando o VRaptor, precisamos que o método retorne a interface `Download`:

```
public Download capa(String isbn) {
```

Existem algumas implementações de `Download` que podemos usar, de acordo com o que temos em mãos. No nosso caso, temos a classe `Arquivo`, que precisamos recuperar usando a `URI` salva no livro:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());
}
```

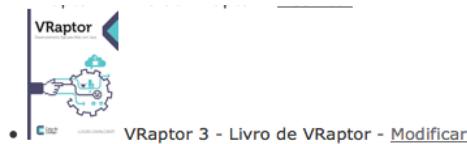
O `Arquivo` nos dá o conteúdo usando um `byte[]`, portanto podemos usar a implementação `ByteArrayDownload` que, além do `byte[]`, também recebe o content type e o nome do arquivo:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());

  return new ByteArrayDownload(capa.getConteudo(),
      capa.getContentType(), capa.getNome());
}
```

Precisamos implementar o método `recupera` do nosso diretório de imagens, recuperando o arquivo, dada a `URI`. Para isso, podemos verificar se a URI é de banco de dados e buscar o `Arquivo` do banco pelo id:

```
@Override
public Arquivo recupera(URI chave) {
  if (chave == null) return null;

  // scheme é o protocolo. No caso de bd:// é o bd
  if (!chave.getScheme().equals("bd")) {
    throw new IllegalArgumentException(chave +
        " não é uma URI de banco de dados");
  }

  // authority é o que vem depois do bd://
  Long id = Long.valueOf(chave.getAuthority());
  return manager.find(Arquivo.class, id);
}
```

Assim, podemos modificar a listagem de livros para incluir a capa:

```
<c:forEach items="${livroList}" var="livro">
    <li>
      <img src="${linkTo[LivrosController].capa[livro.isbn] }"
            width="70" height="100">
      ${livro.titulo} - ${livro.descricao} -
    <a href="${linkTo[LivrosController].edita[livro.isbn] }">
      Modificar
    </a>
  </li>
</c:forEach>
```

Apesar de a tela renderizar sem problemas, se olharmos o log do servidor, veremos vários erros do tipo:

```
Caused by: java.lang.NullPointerException
  at br.com.casadocodigo.livraria.controlador.
      LivrosController.capa(LivrosController.java:89)
```

Isso porque alguns livros não possuem capa, logo, o arquivo retornado pelo DiretorioNoBD vem nulo. Para evitar esses erros, podemos retornar que a imagem não existe no controller, ou seja, retornar um 404:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());

  if (capa == null) {
    result.notFound();
    return null;
  }

  return new ByteArrayDownload(capa.getConteudo(),
      capa.getContentType(), capa.getNome());
}
```

No caso de mudarmos o result, podemos retornar null para bloquear o download. Dessa forma, a página renderiza sem problemas, e o console do servidor fica sem erros.

8.3 Outras implementações de Download

Além da implementação ByteArrayDownload, que consegue gerar um Download a partir de um byte[], temos implementações que usam InputStream e File para gerar o Download. Todas elas possuem um construtor que recebe o conteúdo, o contentType e o nome do arquivo:

```
byte[] capa = //...
return new ByteArrayDownload(capa, "image/png", "capa.png");

InputStream relatorio = //...
return new InputStreamDownload(relatorio, "application/pdf",
    "relatorio.pdf");

File notas = //...
return new FileDownload(notas, "text/plain", "notas.txt");
```

Por padrão, o arquivo é passado para o browser, que pode escolher entre mostrar a caixa de diálogo de download ou simplesmente mostrar o arquivo dentro do browser. Para imagens, textos e pdfs os browsers vão simplesmente mostrar os arquivos na tela. Se quiser forçar o download, todos as implementações de `Download` possuem um construtor que recebe um boolean para forçar o download. Por exemplo, para forçar o download das notas:

```
File notas = //...
return new FileDownload(notas, "text/plain", "notas.txt", true);
```

Se quisermos, podemos implementar `Download` para um tipo específico do nosso sistema. Por exemplo, se usarmos a classe `Arquivo` para representar arquivos em vários modelos do sistema, podemos criar uma implementação `ArquivoDownload`, delegando para o `ByteArrayDownload`:

```
public class ArquivoDownload implements Download {

  private Arquivo arquivo;
  public ArquivoDownload(Arquivo arquivo) {
    this.arquivo = arquivo;
  }

  public void write(HttpServletResponse response) throws IOException {
    Download download = new ByteArrayDownload(arquivo.getConteudo(),
        arquivo.getContentType(), arquivo.getNome());

    download.write(response);
  }
}
```

Em seguida, podemos simplificar o código do controller:

```
@Get("/livros/{isbn}/capa")
public Download capa(String isbn) {
  Livro livro = estante.buscaPorIsbn(isbn);

  Arquivo capa = imagens.recupera(livro.getCapa());

  if (capa == null) {
    result.notFound();
    return null;
```

```
    }
    return new ArquivoDownload(capa);
}
```

Podemos usar essa mesma ideia para gerar relatórios em PDF ou CSV, por exemplo, sem precisarmos nos preocupar com detalhes de como jogar esses arquivos na resposta da requisição, para que o browser faça o download ou simplesmente renderize diretamente o conteúdo.

Capítulo 9

Cuidando da infraestrutura do sistema: Interceptors

Nos capítulos anteriores, vimos ferramentas que nos ajudam no dia a dia do desenvolvimento da nossa aplicação. Usaremos injeção de dependências, controllers, o `Result`, os jsps durante todo o desenvolvimento, a cada nova funcionalidade do sistema. Mas existem algumas tarefas que atingem todo o sistema, ou pelo menos uma grande parte dele. Tarefas como controle de acesso, controle de transações, log de erros etc. Neste capítulo veremos como implementar essas tarefas usando interceptors.

9.1 Executando uma tarefa em vários pontos do sistema: Transações

Quando estamos trabalhando com alguns bancos de dados relacionais, por exemplo, o MySQL InnoDB ou o Oracle, só conseguimos realizar alguma mudança nos dados

se estivermos dentro do ambiente controlado que garante a atomicidade, a consistência, o isolamento e a durabilidade (ACID) da operação: a **Transação**. Consequência disso é que, no nosso sistema, toda vez que quisermos alterar os dados precisamos abrir e fechar (*commitar*) uma transação, para que a alteração realmente aconteça no banco.

Na seção 4.6 resolvemos esse problema pontualmente, no próprio DAO:

```
@Override
public void adiciona(Livro livro) {
  this.manager.getTransaction().begin();
  this.manager.persist(livro);
  this.manager.getTransaction().commit();
}
```

Se seguirmos essa estratégia, a cada método do DAO que altera os dados, teríamos que repetir o código de iniciar transações. Além disso, esse código só está considerando o caminho feliz, em que o dado consegue ser salvo no banco. Transações podem falhar por vários motivos, como violações de restrições do banco (como colunas `NOT NULL`), por referências inválidas, por falhas de conexão etc, portanto precisamos tratar esse caso e fazer o **rollback** da transação.

É um código bastante complicado e que estaria espalhado por todo o sistema. Para evitar essa duplicação de código, vamos pensar no papel do `LivroDAO` no sistema: acessar e modificar os dados de um livro no banco. O controle de transações é uma infraestrutura para que a modificação dos dados ocorra, não faz parte da lógica que o `LivroDAO` deveria executar, não é sua responsabilidade. Além disso, se quisermos adicionar vários livros na mesma transação, já não podemos usar esse código.

Nesse caso, devemos **inverter o controle** e passar o código de controle das transações para uma classe especializada que conseguirá cuidar disso por todo o sistema. No capítulo 4, vimos uma das formas de inversão de controle: injeção de dependências. No entanto, injeção de dependências não vai resolver o nosso problema, pois a transação não é um objeto que podemos receber no construtor, e sim um código que precisa ser executado **em volta** da nossa lógica de negócios: um **aspecto** da nossa aplicação.

Aspecto é todo requisito ou código de infraestrutura que precisamos executar em vários pontos da nossa aplicação, como controle de transação, controle de segurança, logging e auditoria. É um conceito bem importante que até gerou um termo

quando usado: *Programação Orientada a Aspectos*, ou *AOP*. Na prática, não conseguimos desenvolver um sistema inteiro somente usando aspectos mas, nos casos em que eles se aplicam, possibilitam soluções bastante elegantes.

O VRaptor possui uma ferramenta para implementar aspectos: os **Interceptors**. Eles funcionam como os Servlet Filters, possibilitando executar alguma lógica antes e depois da requisição. A diferença é que com os Interceptors podemos receber como dependência qualquer componente da aplicação, então podemos aproveitar o que já está pronto. Podemos pensar que um `Interceptor` vai executar logo antes do método do controller e logo depois desse método.

Para criar um `Interceptor` precisamos implementar a interface do VRaptor com esse nome e anotá-la com `@Intercepts`. No nosso caso, queremos criar um interceptor que cuidará das transações, o `TransacoesInterceptor`, no projeto **livraria-admin**.

```
import br.com.caelum.vraptor.interceptor.Interceptor;
import br.com.caelum.vraptor.Intercepts;

@Intercepts
public class TransacoesInterceptor implements Interceptor {

}
```

Essa interface possui dois métodos: `accepts`, que recebe o objeto do VRaptor que representa um método de Controller — o `ResourceMethod` — e deve retornar um *boolean* indicando se esse método deve ou não ser interceptado; e um método `intercept` que recebe a pilha de interceptors, para podermos continuar a execução do método no momento necessário, ou simplesmente evitar que o método do controller seja executado.

Para o `TransacoesInterceptor`, podemos adotar a solução mais simples para o método `accepts`: retornar `true` e interceptar todos os métodos de controller. Já o código de controle de transação deve estar em volta da chamada ao próximo interceptor da pilha. Para cuidar das transações precisamos de um `EntityManager`, logo, devemos recebê-lo no construtor.

```
@Intercepts
public class TransacoesInterceptor implements Interceptor {

  private EntityManager manager;
```

```
  public TransacoesInterceptor(EntityManager manager) {
    this.manager = manager;
  }

  @Override
  public boolean accepts(ResourceMethod method) {
    return true;
  }

  @Override
  public void intercept(
      InterceptorStack stack,
      ResourceMethod method,
      Object controller) throws InterceptionException {

    //abre a transação antes da execução
    manager.getTransaction().begin();

    //continua a execução do método do controller
    stack.next(method, controller);

    //commita a transação após a execução
    manager.getTransaction().commit();
  }

}
```

Para completar a lógica da transação, precisamos fazer o `rollback` caso haja algum problema. Um dos jeitos de fazer isso é envolvendo o código por um `try..finally` e se, ao final da execução, a transação continuar ativa é porque ela não conseguiu ser *commitada* com sucesso, portanto precisamos fazer o `rollback`:

```
@Override
public void intercept(InterceptorStack stack,
    ResourceMethod method,
    Object controller) throws InterceptionException {

  try {
    manager.getTransaction().begin();

    stack.next(method, controller);
```

```
    manager.getTransaction().commit();
  } finally {
    if (manager.getTransaction().isActive()) {
      manager.getTransaction().rollback();
    }
  }
}
```

Poderíamos também fazer um `catch(Exception e)` nesse bloco e, só então, fazer o `rollback`, mas esse interceptor não tem condição de tratar a exceção e acabaria apenas relançando-a, provavelmente dentro de uma `InterceptionException`, criada especialmente para esse fim. Uma boa regra é, se a classe não vai fazer nada de útil com a exceção além de relançá-la, é melhor nem capturar a exceção, assim podemos passar essa responsabilidade de tratar a exceção para outra classe (um outro interceptor talvez) ou simplesmente deixar aparecer a página de erro 500 para o usuário.

9.2 Controlando os métodos interceptados

Quando criamos o `TransacoesInterceptor`, implementamos o método `accepts` retornando `true`. Isso significa que ele vai interceptar **todos** os métodos de controller, ou seja, todas as requisições web. Mas nem sempre isso é o desejável. Embora essa solução seja suficiente para um sistema pouco acessado, para um sistema com muitos acessos, criar transações a cada requisição pode ser um problema e deixar o sistema mais lento do que deveria.

Transações são necessárias para realizar operações que modificam o banco de dados, mas não são obrigatórias quando fazemos apenas operações de leitura. Nem são necessárias quando estamos apenas mostrando um formulário. Por esse motivo, precisamos implementar o método `accepts` do interceptor para não interceptar os métodos que não precisam de transação.

Analisando a assinatura do método `accepts`, temos o seguinte:

```
public boolean accepts(ResourceMethod method) {...}
```

Recebemos um `ResourceMethod`, que representa o método do controller que está sendo invocado nesta requisição, e precisamos retornar um boolean que diz se o interceptor vai ser usado ou não nesse método. Essa interface nos possibilita acessar:

- O método: `method.getMethod()` retorna um `java.lang.reflect.Method`, que é a classe que representa um método java. Com ele, podemos verificar atributos como nome, parâmetros e anotações presentes no método.
- O controller: `method.getResource().getType()` retorna um `Class<?>`, que é a classe que representa uma classe java. Com ele, podemos verificar atributos como nome, pacote e anotações presentes na classe.

Uma forma de implementar o método `accepts` é listar todos os métodos de controller que queremos interceptar e verificar se o `ResourceMethod` é um deles. Mas essa lista pode ficar bastante extensa na medida em que o projeto vai crescendo, tornando o método `accepts` bem difícil de manter e evoluir.

Uma maneira de melhorar esse registro é criando uma anotação para indicar que o método será interceptado, nesse caso, se o método deverá ser executado dentro de uma transação. Podemos criar a anotação `@Transacional` para esse fim. Para isso, primeiramente devemos criar a classe da anotação, usando a palavra chave `@interface`:

```
public @interface Transacional {

}
```

Em seguida, precisamos dizer em que lugares essa anotação será válida, usando a anotação `@Target`. Os lugares válidos vão de parâmetros e construtores até classes e pacotes. No nosso caso, só precisamos dessa anotação para métodos, então escolhemos esse valor na configuração do `@Target`:

```
@Target(ElementType.METHOD)
public @interface Transacional {

}
```

Outra configuração necessária é a retenção da anotação, ou seja, em que momento ela poderá ser lida. Definimos isso com a anotação `@Retention`, cujos valores possíveis são `SOURCE`, com o qual a anotação só vale como documentação, `CLASS`, com que a anotação pode ser lida durante o processo de compilação, e `RUNTIME`, com o qual a anotação pode ser lida durante a execução da aplicação. Como precisamos ler essa anotação no interceptor, usamos a retenção de `RUNTIME`.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Transacional {

}
```

Agora podemos anotar os métodos desejados, por exemplo, o método `salva` do `LivrosController`:

```
@Transacional
@Post("/livros")
public void salva(Livro livro) {...}
```

E para implementar o método `accepts` podemos usar o método `containsAnnotation` do `ResourceMethod`, passando a anotação `Transacional`:

```
public boolean accepts(ResourceMethod method) {
  return method.containsAnnotation(Transacional.class);
}
```

Assim, todos os métodos de controller que estiverem anotados com `@Transacional` serão interceptados pelo `TransacoesInterceptor` e, portanto, rodarão dentro de transações. Essa é a maneira mais fácil de configurar os métodos interceptados.

No caso específico de transações, podemos usar a nossa interface REST para definir quais métodos podem ser interceptados. Se usamos os métodos HTTP corretamente, sabemos que as requisições `GET` não podem modificar os dados do servidor, então não devem rodar dentro de transações. Por outro lado, as requisições `POST`, `PUT` e `DELETE` supostamente modificam o estado de algum recurso, portanto devem rodar dentro de transação.

Podemos interceptar todos os métodos que não são acessíveis por `@Get`, dispensando a criação de uma nova anotação para esse fim:

```
public boolean accepts(ResourceMethod method) {
  return !method.containsAnnotation(Get.class);
}
```

9.3 Evitando instanciação desnecessária de interceptors: @Lazy

Os interceptors não precisam necessariamente ser aplicados em todas as requisições, mas precisamos instanciá-los para chamar o método `accepts`, justamente para saber se devem ser aplicados. O problema é que essa instanciação faz com que as dependências do interceptor também sejam instanciadas, e as dependências das dependências, e assim sucessivamente.

Esse processo pode ser muito caro para, em alguns casos, nem usar o interceptor. No caso do `TransacoesInterceptor`, que depende de um `EntityManager`, possivelmente estaríamos abrindo uma conexão com o banco de dados para mostrar uma página de formulário que não usa nada do banco.

Para resolver esse problema, podemos usar a anotação `@Lazy`. Com ela, o VRaptor guarda uma instância não funcional do interceptor só para invocar o método `accepts`. Assim, ele só instancia o interceptor caso o `accepts` retorne `true`.

```
@Intercepts @Lazy
public class TransacoesInterceptor implements Interceptor {
  //...
  public boolean accepts(ResourceMethod method) {
     return method.containsAnnotation(Transacional.class);
  }
}
```

Para fazer isso funcionar, o VRaptor instancia o Interceptor passando null para todas as dependências, portanto nem o construtor nem o método `accepts` podem usar as dependências, senão serão lançadas `NullPointerExceptions`. Se conseguirmos usar apenas o método para decidir se o interceptor vai ser aplicado, seja por anotação ou por verificação de outros atributos do método, devemos anotá-lo com `@Lazy`.

Por outro lado, se o interceptor for aplicado em todas as requisições (`accepts` retornando `true` sempre), não há ganho nenhum em usar o `@Lazy`, portanto ele não deve ser usado.

Usando interceptors para redirecionar a requisição

Os interceptors não são obrigados a chamar o `stack.next(method, controller)`, que continua a execução dos interceptors até executar o método do controller. Podemos, por exemplo, fazer um interceptor de segurança, que redireciona para a página de login caso o usuário não esteja logado:

```
@Override
public void intercept(InterceptorStack stack,
    ResourceMethod method,
    Object controller) throws InterceptionException {
  if (usuarioEstaLogado()) {
    stack.next(method, controller);
  } else {
    result.redirectTo(LoginController.class).login();
  }
}
```

Ordem relativa entre interceptors

A ordem em que os interceptors são executados é determinada pela ordem em que eles são lidos pelo VRaptor. Essa ordem não é fixa nem tem regra predefinida. Isso funciona bem na maioria dos casos, em que os interceptors são independentes entre si, mas caso um interceptor precise necessariamente rodar após o outro, é possível usar os atributos `before` e `after` para forçar uma ordem.

```
@Intercepts(before=LoggerInterceptor.class,
            after=ExceptionInterceptor.class)
public class TransacoesInterceptor implements Interceptor {
  ...
}
```

Nesse caso a ordem de execução dos interceptor será: `ExceptionInterceptor -> TransacoesInterceptor -> LoggerInterceptor`.

É possível também forçar a execução dos interceptors em relação aos interceptors do VRaptor que cuidam do processo da requisição. Podemos executar algum código antes ou depois de:

- Definição do método que vai ser executado na requisição: `ResourceLookupInterceptor.class`
- Instanciação do Controller: `InstantiateInterceptor.class`
- Criação dos parâmetros do método: `ParametersInstantiatorInterceptor.class`
- Execução do método do controller: `ExecuteMethodInterceptor.class`

Existem ainda alguns outros interceptors do VRaptor que realizam outras funções, conforme veremos no catálogo de interceptors no capítulo 15.

Outro exemplo de interceptor: Controle de usuários

No nosso sistema, o `livraria-admin` cuidará do cadastros dos livros que vão ser vendidos no site. Esse cadastro pode alterar totalmente como um livro vai ser apresentado no site, portanto só pode ser feito por pessoas qualificadas para tal. Por esse motivo, vamos controlar o acesso de pessoas em geral a esse cadastro, criando uma forma de pessoas se identificarem para realizarem as operações desejadas.

Esse controle de acesso geralmente se dá em dois passos: primeiro, precisamos saber quem é o usuário que está acessando o sistema e, depois, precisamos saber se esse usuário tem permissão de realizar a operação requerida. Chamamos esses dois passos de **Autenticação** e **Autorização**, respectivamente.

A base para esse controle de acesso é criar uma classe que representa o usuário do sistema:

```
public class Usuario {

  private String login;
  private String senha;
  // outros atributos, getters e setters

}
```

Dessa forma, identificamos o usuário que está interagindo com o sistema. Outra parte importante é conseguir saber se o usuário tem permissão para determinadas operações. Esse controle pode ser tão complexo quanto se queira, mas no nosso caso teremos só dois tipos de usuários: admins, que podem alterar os livros e não admins que só podem visualizar os livros. Para isso, vamos acrescentar um campo `admin` na classe `Usuario`:

```
public class Usuario {

  private String login;
  private String senha;

  private boolean admin;
  // outros atributos, getters e setters

}
```

Ao tentar acessar o sistema, precisamos pedir ao usuário que se identifique para continuar, ou seja, que faça o login. Como não queremos ficar pedindo toda requi-

sição para que o usuário se logue, precisamos guardar os seus dados na sessão do usuário, após o login. Para isso, podemos usar um componente de escopo de sessão, representando o usuário logado no sistema:

```
@Component
@SessionScoped
public class UsuarioLogado {

  private Usuario usuario;

  public void loga(Usuario usuario) {
    this.usuario = usuario;
  }

  public boolean isLogado() {
    return this.usuario != null;
  }

  public Usuario getUsuario() {
    return usuario;
  }

  public void desloga() {
    this.usuario = null;
  }
}
```

Dessa forma, podemos criar um controller para apresentar o formulário de login e realizar a autenticação. Esse controller buscará o usuário em algum lugar, como por exemplo, no banco de dados ou no LDAP. Para abstrair o lugar onde ficarão os usuários, vamos usar a interface `RegistroDeUsuarios`. Após encontrar o usuário, vamos guardar as suas informações no `UsuarioLogado`.

```
@Resource
public class LoginController {

  private RegistroDeUsuarios usuarios;
  private UsuarioLogado logado;
  private Result result;

  public LoginController(RegistroDeUsuarios usuarios,
```

```
                             UsuarioLogado logado,
                             Result result,
                             Validator validator) {
    this.usuarios = usuarios;
    this.logado = logado;
    this.result = result;
  }

  @Get("/login")
  public void formulario() {}

  @Post("/login")
  public void login(String login, String senha) {
    Usuario usuario = usuarios.comLoginESenha(login, senha);
    if (usuario == null) {
      validator.add(new I18nMessage("usuario",
                              "login.ou.senha.invalidos"));
    }
    validator.onErrorRedirectTo(this).formulario();

    logado.loga(usuario);

    // ou a página inicial
    result.redirectTo(LivrosController.class).lista();
  }

  @Get("/logout")
  public void logout() {
    logado.desloga();
    result.redirectTo(this).formulario();
  }
}
```

Vou omitir aqui qual vai ser a implementação do `RegistroDeUsuarios`, use a mais apropriada para a sua situação. Vou omitir também o formulário de login, por brevidade.

Com nosso `LoginController` criado, precisamos redirecionar para ele sempre que um usuário tentar acessar o cadastro de livros. Poderíamos ir em cada um dos métodos e verificar se o usuário está logado, mas já temos uma ferramenta para fazer esse tipo de código: os interceptors.

Nesse interceptor, se o usuário não estiver logado, vamos redirecioná-lo para o formulário de login. Para isso vamos precisar do `UsuarioLogado` e do `Result`.

```
@Intercepts
public class AutenticacaoInterceptor implements Interceptor {

  private UsuarioLogado usuario;
  private Result result;

  public AutenticacaoInterceptor(UsuarioLogado usuario, Result result) {
    this.usuario = usuario;
    this.result = result;
  }

  public boolean accepts(ResourceMethod method) {
    return // ???
  }

  public void intercept(InterceptorStack stack,
                        ResourceMethod method,
                        Object controller) {
    if (usuario.isLogado()) {
      stack.next(method, controller);
    } else {
      result.redirectTo(LoginController.class).formulario();
    }
  }
}
```

Falta implementar o método `accepts`. Para isso, poderíamos criar uma anotação `@Restrito`, por exemplo, para identificar as páginas que precisam de login, ou criar uma anotação `@Liberado` para as páginas que não precisam de login, dependendo do número de páginas que serão controladas. No nosso caso, o admin inteiro vai precisar de login, então o `accepts` seria `true`. Mas, assim, até o formulário de login passaria no interceptor e nunca seria executado. Vamos, então, aceitar todos os controllers, menos o de login:

```
public boolean accepts(ResourceMethod method) {
  return !method.getResource().getType().equals(LoginController.class);
}
```

Com isso nosso interceptor de autorização já está pronto. Precisamos, ainda, ver se o usuário logado tem permissão de executar a operação desejada. Para isso, vamos criar outro interceptor, que cuida da autorização. Esse interceptor precisa obrigatoriamente rodar depois do `AutenticacaoInterceptor` e, caso o usuário não tenha permissão, redirecionar para uma página falando isso. O HTTP já tem um status de erro para isso, o `401 -- Unauthorized`, vamos usá-lo.

```
@Intercepts(after=AutenticacaoInterceptor.class)
public class AutorizacaoInterceptor implements Interceptor {

  private UsuarioLogado usuario;
  private Result result;

  public AutorizacaoInterceptor(UsuarioLogado usuario, Result result) {
    this.usuario = usuario;
    this.result = result;
  }

  public boolean accepts(ResourceMethod method) {
    return !method.getResource().getType().
                equals(LoginController.class);
  }

  public void intercept(InterceptorStack stack,
                        ResourceMethod method,
                        Object controller) {
    if (podeAcessar(method)) {
      stack.next(method, controller);
    } else {
      result.use(Results.http()).sendError(401,
                                "Você não está autorizado!");
    }
  }

  private boolean podeAcessar(ResourceMethod method) {
    return // ???
  }
}
```

Precisamos definir a estratégia de autorização, no método `podeAcessar`. Para facilitar, poderíamos criar uma anotação `@Admin` para lógicas que só podem ser

feitas por administradores, ou `@Livre` para lógicas que podem ser acessadas por todos, dependendo da quantidade de lógicas que caem em cada caso. Ou ainda podemos usar uma convenção: tudo que é `@Get` é liberado para todo mundo, pois não modifica nenhum dado (ou pelo menos não deveria), e todo o resto só é acessível por admin.

Para facilitar, vamos adotar essa convenção: o usuário pode acessar o método se ele for `@Get`, ou se ele for `admin`, que pode acessar tudo:

```
public boolean podeAcessar(ResourceMethod method) {
  return method.containsAnnotation(Get.class) ||
    usuario.getUsuario().isAdmin();
}
```

Dessa forma, conseguimos controlar o acesso ao cadastro de livros só para usuários logados, com as modificações aos livros feitos somente por admins. Existem diversas ferramentas que auxiliam esse controle de acessos, como a especificação `JAAS` e o `Spring Security`, que cobrem todos os casos possíveis de autenticação e autorização e já possuem formas padronizadas de fazer autenticação com LDAP, AD e outras tecnologias. Podemos integrá-las de forma transparente ao VRaptor, criando componentes que realizam essa integração.

Podemos também usar os plugins `vraptor-auth` e `SACI-VRaptor`, que podem ser acessados a partir do diretório de plugins do VRaptor: https://github.com/caelum/vraptor-contrib.

Capítulo 10

Melhorando o design da aplicação: Conversores e Testes

Na Orientação a Objetos, temos um objeto como uma abstração que une dados (atributos) a comportamentos (métodos). Mas quando estamos criando uma aplicação que salva dados no banco de dados, tendemos a criar muitas classes que não têm nenhum comportamento: apenas um punhado de atributos com getters e setters, que são apenas acessores desses atributos, e o mapeamento da JPA.

Quando fazemos isso, algumas lógicas ficam espalhadas, muitas vezes em classes `*Util`, quando poderíamos agrupar essas lógicas com os dados em que elas são aplicadas em classes pequenas e especializadas. Veremos neste capítulo como tratar esse tipo de classe e integrá-la facilmente ao processamento da requisição.

Além disso, criar essas classes pequenas melhora o design da aplicação e deveria ser feito constantemente. Uma boa ferramenta para isso é o uso de Testes automatizados, que ajudam a identificar partes do sistema que estão ficando complexas demais e, se usamos o ciclo do Test Driven Development/Design (TDD), os próprios testes

ajudam a melhorar o design da aplicação. Também veremos nesse capítulo como usar TDD em conjunto com os componentes do VRaptor.

10.1 Populando objetos complexos na requisição: Conversores

No capítulo 3 vimos como criar o cadastro de livros, com um formulário capaz de preencher todos os atributos de um `Livro`. Os atributos preenchidos foram:

```
public class Livro {

  private String isbn;
  private String titulo;
  private String descricao;
  private BigDecimal preco;
  private Calendar dataPublicacao;

}
```

Ao criar a classe `Livro`, escolhemos o tipo `BigDecimal` para representar o preço, pois ele consegue representar um valor decimal sem perder precisão. Mas será que esse `BigDecimal` é o suficiente? Imagine que a nossa livraria cresceu e agora vende livros importados. Alguns dos livros virão dos Estados Unidos, com o preço em dólar, outros da Inglaterra, com o preço em libras, e outros ainda da França, com o preço em euro. Agora ao salvar um livro no sistema com o preço valendo `15.90`, como sabemos se ele está em reais, dólares, libras ou euros?

Da maneira que representamos o preço na classe `Livro` não conseguimos saber isso. Precisamos, portanto, guardar o valor da moeda correspondente ao preço do livro. Para isso vamos criar uma enum com as moedas suportadas pelo sistema:

```
public enum Moeda {
  REAL, DOLAR, EURO, LIBRA
}
```

E adicionar um atributo que guarda a moeda na classe `Livro`:

```
public class Livro {
  //...
  private Moeda moeda;
  private BigDecimal preco;
```

```
  //...
}
```

Agora, ao consultarmos o preço do livro, conseguimos saber qual é a moeda olhando para o atributo `moeda`. Se quisermos somar o valor de vários livros, por exemplo, num carrinho de compras, precisaremos considerar sempre o preço e a moeda. Toda a vez que olharmos para o preço do livro precisaremos olhar também para a moeda e se, em algum lugar do código, esquecermos de fazer isso, podemos somar um preço em reais com um preço em euros e chegar num valor que não é nem uma coisa nem outra!

Apesar de termos guardado a moeda na classe `Livro`, moeda não é um atributo do `Livro`, é um atributo do preço! Assim como não temos um atributo `mes` no livro e sim uma data de publicação que possui um mês, deveríamos ter um atributo `preco` no livro que, além de guardar o valor do preço, guardaria também a moeda. Para fazer isso, precisamos de uma classe que agrupa esses dois valores, a classe `Dinheiro`:

```
public class Dinheiro {
  private Moeda moeda;
  private BigDecimal montante;
}
```

E o preço da classe `Livro` passa a ser desse tipo:

```
public class Livro {

   //...
   //-private Moeda moeda;
   private Dinheiro preco;

}
```

Essa classe `Dinheiro` é uma classe que guarda dados mas não é uma classe que salvaríamos no banco de dados como uma tabela. É uma classe que guarda um **valor**, nesse caso um valor em dinheiro, como por exemplo `R$ 29,90`. Dois objetos do tipo `Dinheiro` que têm a mesma `moeda` e o mesmo `montante` guardam exatamente o mesmo valor. Classes desse tipo são os chamados `Value Objects` (http://martinfowler.com/bliki/ValueObject.html) , ou seja, classes que representam valores.

Ao criar a classe Dinheiro, podemos concentrar todas as operações monetárias dentro dela, como somar preços ou converter um preço de real para dólar, e podemos fazer verificações adicionais, como não permitir somar preços de moedas diferentes.

Para popular o preço do livro no nosso sistema agora precisaríamos de dois campos no formulário:

```
<li>Moeda: <br/>
  <input type="text" name="livro.preco.moeda"
      value="${livro.preco.moeda}"/></li>
<li>Montante: <br/>
  <input type="text" name="livro.preco.montante"
      value="${livro.preco.montante}"/></li>
```

Mas o livro não possui moeda e montante, ele possui um preço! O ideal é que o usuário possa digitar no campo de preço algo como "R$ 52,00" e o sistema conseguir salvar o preço correspondente. Mas como fazer isso? Antes, vamos analisar os atributos da classe Livro.

```
public class Livro {

  private String isbn;
  private String titulo;
  private String descricao;
  private Calendar dataPublicacao;
  private Dinheiro preco;

}
```

Os primeiros atributos são Strings e, para serem populados, basta pegar o parâmetro que vem da requisição, ou seja, o valor digitado no input. Neste caso, não importa o que foi digitado no input, os campos String serão populados com esse valor.

No entanto, os dois últimos valores são mais complexos: o preco, que é um Dinheiro, e a dataPublicacao, que é um Calendar. Quando queremos popular a data de publicação, digitamos um valor no campo correspondente do formulário, por exemplo, 17/06/2013. Esse valor vai para o servidor como a String "17/06/2013", mas para poder passar esse valor para a classe Livro, o VRaptor precisa converter "17/06/2013" num Calendar que representa esse dia. Precisamos que essa mesma lógica se aplique quando digitarmos R$ 29,90 no campo

"Preço" do formulário, queremos que o VRaptor converta para o `Dinheiro` com moeda `REAL` e montante `29.90`.

Isso é feito através de classes especializadas em converter os parâmetros da requisição em objetos que representam esses valores. Essas classes são `Converter`s, ou seja, implementam a interface `Converter` do VRaptor e são responsáveis por converter uma `String` que veio da requisição em um tipo específico.

Mapeando a classe Dinheiro no Hibernate

Para mapear a classe `Dinheiro`, não vamos criar uma tabela separada, já que `Dinheiro` é um *Value Object*. Ao invés disso vamos mapeá-la dentro da tabela `Livro`, usando a anotação `@Embedded`:

```
@Entity
public class Livro {
  //...
  @Embedded
  public Dinheiro preco;
}
```

Assim, serão criadas duas colunas na tabela Livro, a `moeda` e o `montante`. Além disso, é necessário criar um construtor sem argumentos dentro da classe `Dinheiro`, mesmo que seja `protected` para que o hibernate consiga instanciar essa classe a partir dos dados do banco.

O VRaptor possui um conjunto de Converters já implementados, todos no pacote `br.com.caelum.vraptor.converter`, e são o suficiente para popular todos os tipos simples do Java: `String`, números, `boolean`, `enum` e datas. Esses conversores padrão são suficientes para a maioria dos objetos que populamos na requisição, mas precisaremos criar um personalizado para a nossa classe `Dinheiro`. Para isso, devemos criar uma classe anotada `@Convert(Dinheiro.class)` que implementa `Converter<Dinheiro>`.

```
import br.com.caelum.vraptor.Convert;
import br.com.caelum.vraptor.Converter;

@Convert(Dinheiro.class)
public class DinheiroConverter
```

```
        implements Converter<Dinheiro> {

    @Override
    public Dinheiro convert(
        String value,
        Class<? extends Dinheiro> type,
        ResourceBundle bundle) {
      return null;
    }

}
```

A interface `Converter` define um único método que deve pegar a String passada e converter no tipo desejado, no caso o `Dinheiro`. Para auxiliar a conversão, existem outros dois parâmetros, o `type` que é a classe do tipo de destino, para ser usado caso estejamos convertendo para implementações de uma interface ou filhos de uma classe, e o `bundle` para gerar mensagens de erro internacionalizadas.

Para facilitar a implementação desse `Converter`, vamos usar TDD — *Test Driven Design*, criando a classe `DinheiroConverterTest` dentro de `src/main/test`. Para iniciar o ciclo do TDD, vamos criar um teste que falha, usando JUnit 4:

```
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;

public class DinheiroConverterTest {
  @Test
  public void converteUmValorEmReais() {
    Converter<Dinheiro> converter = new DinheiroConverter();

    assertThat(converter.convert("R$ 1,00", null, null),
            is(new Dinheiro(Moeda.REAL, new BigDecimal("1.00"))));
  }
}
```

Onde aprender TDD?

Recomendo fortemente que você aprenda TDD e adote a prática nos sistemas que desenvolver. Definitivamente ele aumenta a qualidade do software escrito. Um excelente material é o livro escrito pelo Mauricio Aniche, e disponível no site da Editora Casa do Código — http://www.casadocodigo.com.br.

Para testar, estamos usando a DSL (*Domain Specific Language*) do JUnit e os `Matchers` do Hamcrest que vimos no capítulo de validações (6.2). Nesse teste estamos garantindo que a conversão do valor `"R$ 1,00"` seja o `Dinheiro` em real no valor de `1.00`. Usando essa DSL conseguimos ler essa frase anterior, em inglês, ou seja, temos um teste bastante legível. Ao rodar o teste (no Eclipse, botão direito > Run As > JUnit Test) vemos que ele falha:

```
java.lang.AssertionError:
Expected: is <br.com.casadocodigo.livraria.modelo.Dinheiro@692a3722>
     got: null
```

Essa saída fala que esperava um `Dinheiro` específico, mas veio `null`. O problema é que apenas usando essa mensagem não conseguimos saber qual é o valor em dinheiro que estamos esperando. Para corrigir isso vamos implementar o `toString` da classe `Dinheiro`. O formato desejado, para facilitar a leitura, pode ser `"Dinheiro(R$ 1.00)"`. Para não precisarmos ficar fazendo um monte de `ifs`, podemos modificar a `enum` de `Moeda` para incluir o símbolo de cada constante:

```java
public enum Moeda {
  REAL("R$"), DOLAR("US$"),
  EURO("€"), LIBRA("£");

  private final String simbolo;

  private Moeda(String simbolo) {
    this.simbolo = simbolo;
  }

  public String getSimbolo() {
```

```
    return simbolo;
  }
}

public class Dinheiro {
  //...
  @Override
  public String toString() {
    return String.format("Dinheiro(%s %s)",
        moeda.getSimbolo(),
        montante
      );
  }
}
```

Agora, ao rodarmos o teste, veremos a seguinte mensagem:

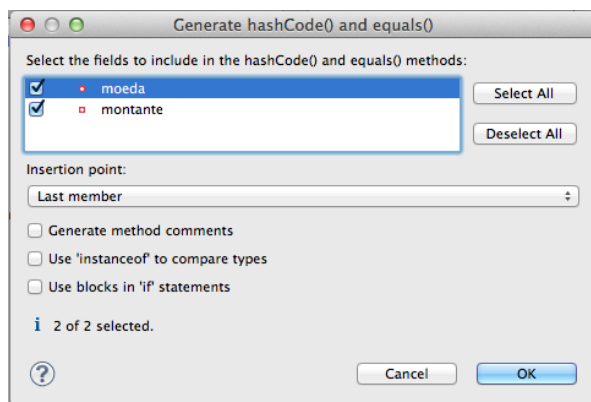
```
java.lang.AssertionError:
Expected: is <Dinheiro(R$ 1.00)>
     got: null
```

Para fazer o teste passar rapidamente, vamos simplesmente retornar o `Dinheiro` desejado:

```
@Convert(Dinheiro.class)
public class DinheiroConverter
      implements Converter<Dinheiro> {

  @Override
  public Dinheiro convert(
      String value,
      Class<? extends Dinheiro> type,
      ResourceBundle bundle) {

    return new Dinheiro(Moeda.REAL, new BigDecimal("1.00"));
  }

}
```

Agora, se rodarmos o teste, ele ainda falha, mas desta vez com essa mensagem:

```
java.lang.AssertionError:
Expected: is <Dinheiro(R$ 1.00)>
     got: <Dinheiro(R$ 1.00)>
```

Ora, esperamos um `Dinheiro` de `R$ 1.00` e apareceu um `Dinheiro` de `R$ 1.00`. O que deu errado? Apesar de guardarem o mesmo valor, são dois objetos diferentes na memória, por isso o teste falha. Nossa classe `Dinheiro` é um *Value Object*, que é definida unicamente pelos seus atributos. Ou seja, um dinheiro é igual ao outro se tem a mesma moeda e o mesmo montante. E para dizer isso, precisamos implementar o `equals` e o `hashCode` da classe `Dinheiro`, de preferência gerando esses métodos com a ajuda da sua IDE. No Eclipse, usamos o menu *Source* > *Generate hashCode() and equals()* e selecionamos os dois atributos.

Agora, ao rodarmos novamente o teste, ele passa! Completamos a segunda parte do ciclo do TDD, fazendo o teste passar e devemos passar para a próxima parte: a Refatoração. Claro que retornar direto o Dinheiro de R$ 1.00 não é uma implementação razoável, então vamos modificá-la para converter valores em Real. Primeira tentativa vai ser, se a string começar por `"R$"`, tentar criar um `BigDecimal` a partir dela.

```
public Dinheiro convert(String value, ...) {
  if (value.startsWith("R$")) {
    return new Dinheiro(Moeda.REAL,
      new BigDecimal(value.replace("R$ ", ""))
    );
  }
  return null;
}
```

Porém, ao rodarmos o teste, recebemos um erro:

```
java.lang.NumberFormatException
  at java.math.BigDecimal.<init>(BigDecimal.java:459)
  at java.math.BigDecimal.<init>(BigDecimal.java:728)
  at br.com.casadocodigo.livraria.conversor.DinheiroConverter
        .convert(DinheiroConverter.java:23)
```

Isso aconteceu porque o construtor do `BigDecimal` espera os números no formato `1.00`, e não `1,00` como foi passado. Para resolver isso, vamos trocar toda vírgula por ponto:

```
@Convert(Dinheiro.class)
public class DinheiroConverter
      implements Converter<Dinheiro> {

  @Override
  public Dinheiro convert(
      String value,
      Class<? extends Dinheiro> type,
      ResourceBundle bundle) {

    if (value.startsWith("R$")) {
      return new Dinheiro(Moeda.REAL,
        new BigDecimal(
          value.replace("R$ ", "")
               .replace(',', '.')
        )
      );
    }
    return null;
  }
}
```

Ao rodarmos novamente o teste, vemos que ele passa e, se estivermos satisfeitos com a implementação, completamos a terceira parte do ciclo do TDD. Como é um ciclo, precisamos voltar para a primeira parte e escrever um teste que falha. Podemos escrever um teste para converter valores em outra moeda, por exemplo em Dólar.

```
@Test
public void converteUmValorEmDolares() {
  Converter<Dinheiro> converter = new DinheiroConverter();
```

```
  assertThat(converter.convert("US$ 49,95", null, null),
      is(new Dinheiro(Moeda.DOLAR, new BigDecimal("49.95"))));
}
```

Ao rodarmos o teste, ele falha com a mensagem:

```
java.lang.AssertionError:
Expected: is <Dinheiro(US$ 49.95)>
     got: null
```

A mensagem está clara o suficiente, então vamos modificar a implementação para fazer o teste passar. Uma implementação mais direta é copiar o if de reais e transformá-lo em dólar:

```
if (value.startsWith("R$")) {
  return new Dinheiro(Moeda.REAL,
    new BigDecimal(
        value.replace("R$ ", "")
             .replace(',', '.')
    )
  );
} else if (value.startsWith("US$")) {
  return new Dinheiro(Moeda.DOLAR,
    new BigDecimal(
        value.replace("US$ ", "")
             .replace(',', '.')
    )
  );
}
return null;
```

Ao rodarmos os testes, vemos que tanto o teste de reais quanto o de dólares passam. Podemos então, passar para a refatoração. Se continuarmos com o código dessa maneira teremos um if para cada moeda, com um código bem parecido dentro de cada `if`. As únicas coisas que mudam são a moeda e o símbolo, ambos acessíveis pelas constantes da enum `Moeda`. Vamos modificar, então, para passar por todas as constantes da `enum` e testá-las:

```
for (Moeda moeda : Moeda.values()) {
  if (value.startsWith(moeda.getSimbolo())) {
    return new Dinheiro(moeda,
```

```
        new BigDecimal(
          value.replace(moeda.getSimbolo(), "")
               .replace(',', '.').trim()
        )
    );
  }
}

return null;
```

Para manter o comportamento anterior, adicionamos um `trim()` para remover os espaços em branco do começo da `String`. Ao rodar os testes vemos que continuam passando, então completamos outro ciclo. Para começar outro ciclo poderíamos testar outra moeda, mas o teste não iria falhar, já que estamos contemplando todas as moedas no converter.

Os casos felizes (*happy paths*) já foram implementados, agora precisamos pensar nos casos patológicos, por exemplo se passam um valor que não é um dinheiro. Do jeito que está a implementação ele vai converter para `null` nesse caso, mas se deixarmos assim podemos mascarar erros. Quando a `String` que tentamos converter não é conversível para o tipo desejado, precisamos sinalizar com um erro. A API de converters do VRaptor já possui um erro implementado para isso, o `ConversionError`. Esse erro já é automaticamente convertido numa mensagem de validação. Podemos mostrá-lo no formulário sem nenhum código adicional.

Vamos, então, começar outro ciclo do TDD, criando um teste que espera o erro de conversão:

```
@Test(expected=ConversionError.class)
public void lancaErroDeConversaoQuandoOValorEhInvalido() {
  Converter<Dinheiro> converter = new DinheiroConverter();
  converter.convert("noventa pratas!", null, null);
}
```

Ao rodar o teste, recebemos a mensagem:

```
java.lang.AssertionError:
Expected exception: br.com.caelum.vraptor.converter.ConversionError
```

Para fazer o teste passar, precisamos lançar esse erro, caso não seja possível converter:

```
public Dinheiro convert(String value, ...) {
  for(Moeda moeda : Moeda.values()) {...}

  throw new ConversionError(value + " não é um dinheiro válido");
}
```

Ao rodarmos os testes, vemos que eles passam. Para melhorar a geração das mensagens, podemos usar o parâmetro `bundle` que recebemos no método `convert`, para podermos internacionalizar a mensagem. Para isso, podemos fazer:

```
throw new ConversionError(
  MessageFormat.format(bundle.getString("dinheiro_invalido"), value)
);
```

E colocar a chave `"dinheiro_invalido"` no `messages.properties`. Ao rodarmos o teste recebemos uma `NullPointerException`, pois não passamos o bundle. Vamos corrigir isso mudando o teste para usar o bundle de mensagens:

```
@Test(expected=ConversionError.class)
public void lancaErroDeConversaoQuandoOValorEhInvalido() {
  Converter<Dinheiro> converter = new DinheiroConverter();
  converter.convert("noventa pratas!", null,
      ResourceBundle.getBundle("messages"));
}
```

Ao rodar o teste, ele ainda falha, com a mensagem:

```
java.lang.Exception:
Unexpected exception,
  expected<br.com.caelum.vraptor.converter.ConversionError>
  but was<java.util.MissingResourceException>
  ...
Caused by: java.util.MissingResourceException:
  Can't find resource for bundle java.util.PropertyResourceBundle,
  key dinheiro_invalido
  ...
```

Ou seja, ele esperava o `ConversionError`, mas aconteceu uma `MissingResourceException`, que diz que a chave `dinheiro_invalido` não foi encontrada no bundle passado. Basta adicioná-la no `messages.properties`:

```
dinheiro_invalido = "{0}" não é um dinheiro válido
```

Agora sim, ao rodarmos os testes vemos que todos eles passam. Outro caso em que vai falhar é quando a `String` começa com um símbolo, mas o número é inválido:

```
@Test(expected=ConversionError.class)
public void lancaErroDeConversaoQuandoOMontanteEhInvalido() {
  Converter<Dinheiro> converter = new DinheiroConverter();
  converter.convert("R$ mil", null,
                    ResourceBundle.getBundle("messages"));
}
```

A mensagem de erro ao rodar o teste:

```
java.lang.Exception:
  Unexpected exception,
    expected<br.com.caelum.vraptor.converter.ConversionError>
    but was<java.lang.NumberFormatException>
...
Caused by: java.lang.NumberFormatException
...
```

Para corrigir precisamos envolver a criação do `BigDecimal` num `try..catch` e lançar um `ConversionError` em vez de um `NumberFormatException`:

```
for (Moeda moeda : Moeda.values()) {
  if (value.startsWith(moeda.getSimbolo())) {
    try {
      return new Dinheiro(moeda,
          new BigDecimal(
            value.replace(moeda.getSimbolo(), "")
              .replace(',', '.').trim()
          )
      );
    } catch (NumberFormatException e) {
      throw new ConversionError(
        MessageFormat.format(bundle.getString("dinheiro_invalido"),
          value)
      );
    }
  }
}
throw new ConversionError(
```

```
  MessageFormat.format(bundle.getString("dinheiro_invalido"), value)
);
```

Essa implementação faz os testes passarem, mas começou a ficar ruim. Vamos, portanto, refatorá-la, para completar o ciclo do TDD. Para isso vou extrair a criação do `BigDecimal` para um método, levando junto o `try..catch`:

```
@Override
public Dinheiro convert(
    String value,
    Class<? extends Dinheiro> type,
    ResourceBundle bundle) {

  for (Moeda moeda : Moeda.values()) {
    if (value.startsWith(moeda.getSimbolo())) {
      return new Dinheiro(moeda, criaMontante(value, moeda, bundle));
    }
  }
  throw new ConversionError(
    MessageFormat.format(bundle.getString("dinheiro_invalido"), value)
  );

}

private BigDecimal criaMontante(
    String value,
    Moeda moeda,
    ResourceBundle bundle) {
  try {
    return new BigDecimal(
      value.replace(moeda.getSimbolo(), "")
        .replace(',', '.').trim()
    );
  } catch (NumberFormatException e) {
    throw new ConversionError(
      MessageFormat.format(bundle.getString("dinheiro_invalido"),
        value)
    );
  }
}
```

Rodando os testes, eles continuam passando e poderíamos continuar refatorando, mas vamos parar por aqui. Um último caso que a gente não cobriu é quando o usuário não preencheu o valor de dinheiro. Nesse caso a `String` vai vir vazia e, do jeito que está a implementação, vai lançar um `ConversionError`. Não é a resposta mais adequada, melhor seria retornar `null`, já que o campo não foi preenchido. Vamos, então, adicionar um teste para isso:

```
@Test
public void converteStringVaziaEmNull() {
  Converter<Dinheiro> converter = new DinheiroConverter();
  ResourceBundle bundle = ResourceBundle.getBundle("messages");

  assertThat(converter.convert("", null, bundle),
      is(nullValue()));
}
```

Esse teste falha com a mensagem:

```
br.com.caelum.vraptor.converter.ConversionError:
  "" não é um dinheiro válido
```

Para fazer o teste passar, vamos colocar essa condição de string vazia no começo do método `convert`:

```
public Dinheiro convert(String value, ...) {
  if (Strings.isNullOrEmpty(value)) { return null; }

  for (Moeda moeda : Moeda.values()) { ... }

  throw new ConversionError(...);
```

Os testes passam e esgotamos os casos que queremos suportar no converter. Com o apoio dos testes, agora conseguimos refatorar a classe `DinheiroConverter` sem medo. Fazer testes durante o desenvolvimento é essencial para que tenhamos segurança para evoluir o sistema sem quebrar comportamentos existentes. Além disso, ao usarmos o ciclo do TDD conseguimos criar classes com um design melhor, mais simples de usar e, mais ainda, ganhamos uma documentação viva sobre o que a classe faz: basta ler os casos de teste. Para saber mais sobre TDD, existe o livro "Test Driven Design em Java" por Maurício Aniche, pela Casa do Código, além das referências, em inglês, "TDD by Example" do Kent Beck e "Growing Object Oriented Software Guided by Tests" de Nat Pryce.

10.2 Testes de unidade em projetos que usam VRaptor

Na seção anterior, vimos como criar testes usando TDD para a classe `DinheiroConverter`. Esse tipo de testes deveria ser feito para todas as classes do sistema, para garantir que cada unidade (classe, método, funcionalidade) funcione isoladamente. Como aplicar isso para as classes de um projeto que usa VRaptor?

Um dos grandes pilares do VRaptor é a injeção de dependências, que vimos em detalhes no capítulo 4. Por causa disso, a maioria das classes do sistema não depende de nada do VRaptor: são componentes (`@Component`) que dependem de outros componentes da aplicação. Se juntarmos isso à pratica de sempre depender de interfaces, temos classes independentes das implementações das suas dependências.

Com isso, ganhamos de graça uma melhor testabilidade das classes. Por exemplo, para testarmos a classe `AcervoNoAdmin` que criamos dentro do `livraria-site`, temos como dependência um `ClienteHTTP`:

```
public class AcervoNoAdmin implements Acervo {
  private ClienteHTTP http;

  public AcervoNoAdmin(ClienteHTTP http) {
    this.http = http;
  }

  public List<Livro> todosOsLivros() {...}
}
```

Essa classe faz uma requisição usando esse cliente HTTP e espera um resultado em xml, que é convertido para uma lista de livros. Para testar essa classe isoladamente, podemos passar uma implementação falsa da interface ClienteHTTP que retorna um xml determinado:

```
@Test
public void converteUmaListaComApenasUmLivro() {
  ClienteHTTP http = new ClienteHTTP() {
    @Override
    public String get(String url) {
      return
        "<livros>" +
```

```
          "<livro>" +
            "<titulo>VRaptor 3</titulo>" +
            "<isbn>12345</isbn>" +
          "</livro>" +
        "</livros>";
    }
  };
  AcervoNoAdmin acervo = new AcervoNoAdmin(http);

  List<Livro> livros = acervo.todosOsLivros();

  assertThat(livros.size(), is(1));

  Livro livro = livros.get(0);
  assertThat(livro.getTitulo(), is("VRaptor 3"));
  assertThat(livro.getIsbn(),   is("12345"));
}
```

Para evitar a criação de classes para cada cenário de testes, podemos usar frameworks de **Mocks**, um dos nomes dados a essas implementações falsas usadas em testes. Um desses frameworks é o **Mockito** (http://mockito.org) , que simplificaria o teste para:

```
import static org.mockito.Mockito.*;

//...

@Test
public void converteUmaListaComApenasUmLivro() {
  ClienteHTTP http = mock(ClienteHTTP.class);
  when(http.get(anyString())).thenReturn(
      "<livros>" +
        "<livro>" +
          "<titulo>VRaptor 3</titulo>" +
          "<isbn>12345</isbn>" +
        "</livro>" +
      "</livros>"
  );
  //...
}
```

Seguindo essa ideia, podemos fazer o teste da maioria dos componentes da apli-

cação, usando qualquer técnica de testes para objetos em geral. A exceção disso são os Controllers, que geralmente dependem de `Result` e `Validator`. Essas interfaces são um pouco mais difíceis de mockar, pois têm uma interface fluente:

```
result.redirectTo(LoginController.class).formulario();
validator.onErrorRedirectTo(this).lista();
```

O mock de uma dessas chamadas, usando o mockito, seria algo como:

```
LoginController mockController = mock(LoginController.class);

when(result.redirectTo(LoginController.class)).
            thenReturn(mockController);

verify(mockController).formulario();
```

Se não fizermos pelo menos as duas primeiras linhas, a execução falha com uma `NullPointerException`. Se estivermos usando outro tipo de resultado, como xml ou json, a configuração dos mocks fica ainda mais difícil. Para evitar isso, podemos usar as versões mockadas providenciadas pelo VRaptor, que já ignoram as chamadas do result e nos deixam inspecionar alguns tipos de dados. Esses mocks do VRaptor ficam no pacote `br.com.caelum.vraptor.util.test` e os principais são:

- `MockResult` — um result que ignora os redirecionamentos e o método `use`, enquanto possibilita a inspeção de todos os objetos incluídos:

  ```
  MockResult result = new MockResult();

  //dentro do controller:
  result.include("mensagem", "Tudo deu certo!");

  //dentro do teste:
  assertThat(result.included("mensagem"), is("Tudo deu certo!"));
  ```

- `MockValidator` - um validator que acumula as mensagens adicionadas e lança exceção no `validator.onErrorUse()`. Assim, podemos verificar que houveram erros de validação:

  ```
  MockValidator validator = new MockValidator();
  LivrosController controller = new LivrosController(..., validator);
  ```

```
Livro livroSemTitulo = //...

try {
  controller.salva(livroSemTitulo);
  Assert.fail("Deveria ter dado erro de validação");
} catch (ValidationException e)
  assertThat(e.getErrors().size(), is(1));
  Message message = e.getErrors().get(0);
  assertThat(message.getCategory(), is("titulo"));
}
```

- `MockSerializationResult` — igual ao `MockResult`, mas também permite inspecionar os dados de um objeto serializado:

```
MockSerializationResult result = new MockSerializationResult();

//no controller
result.use(Results.json()).from(livro).serialize();

//no teste
assertThat(result.serializedResult(), is(
  "{\"livro\": { \"titulo\": \"VRaptor 3\", ...  }}"
));
```

- `JSR303MockValidator` — igual ao `MockValidator`, mas também executa as validações do BeanValidations (chamadas ao `validator.validate(objeto)`).

Desse modo, conseguimos também testar os controllers sem muitos problemas e manter uma boa cobertura de testes pela aplicação.

Capítulo 11

Próximos passos

Terminamos, aqui, o desenvolvimento da nossa livraria, que foi o projeto que usamos para acompanhar esse livro e expôr a maioria das funcionalidades do VRaptor. Mas não podemos parar por aqui, pois cada projeto que fazemos tem características diferentes, resolve problemas diferentes, exige coisas diferentes.

Para continuar a conhecer mais sobre o VRaptor, podemos seguir os seguintes passos:

- **Leia os apêndices deste livro**. Nos apêndices, temos uma visão geral dos plugins do VRaptor, que auxiliam o desenvolvimento de aplicações com o VRaptor, uma exposição de todos os containers de injeção de dependências suportados pelo VRaptor, exemplos de como usar AJAX com o VRaptor e um pouco sobre o funcionamento interno do VRaptor, com seus componentes e suas funções;

- **Leia a documentação do VRaptor**. Apesar de não estar completa, a documentação do VRaptor, que pode ser encontrada em http://vraptor.caelum.

com.br é uma boa referência para vermos as funcionalidades do VRaptor. Além da documentação existe o livro de receitas, com a solução para alguns problemas comuns do dia-a-dia;

- **Contribua com a documentação do VRaptor**. Colocar no papel (mesmo que seja digital) o que você conhece ajuda a consolidar o seu conhecimento e ajudar aos outros. Portanto, escreva um blog técnico com os problemas e soluções que você adotou, contribua com uma solução para o livro de receitas do VRaptor, ajude a melhorar a documentação existente, em https://github.com/caelum/vraptor/tree/gh-pages;

- **Participe da lista de discussões do VRaptor**. Cada vez que aparecer uma dúvida ou um problema durante o desenvolvimento das suas aplicações, ou mesmo se você tiver sugestões ou quiser chamar uma discussão, acompanhe e mande mensagens para a lista de discussões caelum-vraptor@googlegroups.com. Tentar resolver o problema de outras pessoas também ajuda bastante a conhecer novas formas de trabalhar com o VRaptor e aprender cada vez mais;

- **Crie e responda tópicos no GUJ**. O guj.com.br é o maior forum de Java em português, então sempre que tiver uma dúvida que envolva, não só o VRaptor, mas qualquer biblioteca que você estiver usando na aplicação, você pode usar o conhecimento das centenas de milhares de usuários do GUJ. Pesquise se já existe um tópico que responde a sua dúvida e, se não existir, crie um novo tópico. Não se esqueça de postar stacktraces, caso estejam acontecendo erros e, se possível, poste o pedaço do código onde você acha que esteja acontecendo o problema. Se o tópico é relacionado ao VRaptor, use o fórum de *Ferramentas e bibliotecas brasileiras*. Não se esqueça de retribuir o favor e responder a dúvida de outras pessoas!

- **Crie e responda perguntas no GUJ**. O guj.com.br possui um modo de perguntas e respostas, o http://www.guj.com.br/perguntas que também é bastante interessante para obtermos a resposta para nossas dúvidas. Seja encontrando uma pergunta já existente ou criando uma nova. Ganhe pontos e o respeito de outros desenvolvedores respondendo perguntas!

- **Sugira novas funcionalidades e reporte bugs**. A sua opinião é muito importante para o desenvolvimento do VRaptor, então sempre que sentir falta de

alguma funcionalidade ou encontrar um bug no VRaptor, mande uma mensagem na lista de discussões do desenvolvimento do próprio VRaptor, a caelum-vraptor-dev@googlegroups.com, para ver se a funcionalidade já existe, seja no VRaptor ou em algum de seus plugins, e se o bug possui alguma forma de ser contornado. Se preferir, pode abrir um ticket de nova funcionalidade ou de bug diretamente no repositório do VRaptor, em https://github.com/caelum/vraptor/issues;

- **Contribua desenvolvendo para o VRaptor**. O código fonte do VRaptor está no Github, que é, sem dúvida, a melhor ferramenta de compartilhamento de código que temos atualmente. Uma ótima forma de entender melhor o VRaptor é olhando o seu código fonte, em https://github.com/caelum/vraptor. O VRaptor possui uma ótima suite de testes, que exemplificam muito bem o uso de cada funcionalidade, além de garantir que elas estão funcionando. Com isso, além de aprender mais sobre o VRaptor, também serve de referência sobre formas de escrever testes em Java. Quando se sentir confortável (não hesite em postar dúvidas sobre o código do VRaptor na lista de desenvolvimento caelum-vraptor-dev@googlegroups.com), escolha algum dos tickets em https://github.com/caelum/vraptor/issues e implemente-o, abrindo um Fork no Github e mandando Pull Requests. Se você já tem uma ideia de funcionalidade ou já encontrou um bug, pode mandar Pull Requests mesmo que eles não tenham sido reportados ainda. Durante a revisão do seu Pull Request, você ainda pode ganhar dicas de design, de como escrever testes, de como contribuir melhor para projetos open-source;

- **Escreva e divulgue plugins**. Conforme visto no apêndice de plugins, o VRaptor possui um mecanismo que facilita a criação e a distribuição de plugins. Então se você criou um conjunto de componentes que pode ser reutilizado para outras aplicações, crie um plugin e divulgue-o no repositório de plugins: https://github.com/caelum/vraptor-contrib.

- **Continue lendo e estudando**. Mantenha-se atualizado e seja um desenvolvedor completo. Conhecer as técnicas para implementar um bom back-end é essencial, mas saber como fazer um front-end agradável para sua aplicação pode ser um grande diferencial e claro, o tornar um desenvolvedor com muito mais conhecimento e domínio do seu trabalho. A Editora Casa do Código tem diversos outros livros que cobrem tanto o back-end como o front-end. Não deixe de ver se é o caminho adequado para continuar seus estudos.

Muito obrigado por ler este livro, espero que ele tenha te ajudado a conhecer ou se aprofundar no VRaptor. O sua opinião é muito importante, tanto para este livro quanto para o próprio VRaptor, então se quiser nos dar seu feedback use a lista de discussão caelum-vraptor@googlegroups.com.

Muito obrigado, também, a você que contribuiu para o VRaptor, seja escrevendo código, documentação, respondendo dúvidas de outras pessoas, divulgando plugins, ou simplesmente usa e recomenda o VRaptor para outras pessoas.

Agradecimentos especiais aos irmão Paulo e Guilherme Silveira, por criarem e incentivarem o VRaptor e por me chamarem para trabalhar na Caelum quando eu era apenas um estudante de Ciências da Computação, ao Fabio Kung, por ser o grande evangelista do VRaptor 2, ao Otávio Garcia por ter me ajudado a coordenar o desenvolvimento do VRaptor nos últimos anos, à Ceci Fernandes, por aguentar minhas piadas ruins desde o tempo da faculdade, ao Pedro Matiello, pelo mesmo motivo e por ajudar a revisar este livro, ao Adriano Almeida, por me deixar escrever esse livro, à minha família e à minha (futura) esposa Melissa por todo o apoio.

=)

Capítulo 12

Apêndice: Melhorando a usabilidade da aplicação com AJAX

Foi-se o tempo em que os usuários apenas seguiam links e submetiam formulários na Web. Hoje em dia, todo mundo espera que uma aplicação Web tenha uma interface rica, com componentes inteligentes e interações suaves. Isso porque transferimos grande parte das aplicações que eram tipicamente Desktop para a Web e precisamos manter, e mesmo melhorar, a usabilidade dessas aplicações.

Por um tempo se supriu essa necessidade com pequenas aplicações em Flash, que eram bastante lentas e tinham músicas e animações totalmente desnecessárias, com o famoso "pular introdução". Agora que a maioria da internet não usa mais o famigerado Internet Explorer 6, temos condição de criar interfaces ricas usando apenas o que o browser nos dá: HTML, CSS e Javascript, com a ajuda de algumas bibliotecas Javascript, é claro.

Uma das principais bibliotecas Javascript é o jQuery, que possui diversos plugins e componentes visuais que facilitam muito o trabalho de um desenvolvedor Web.

Além disso, existem bibliotecas para facilitar a escrita do CSS como o LESS ou o SASS. Para escrever HTML em Java temos o JSP com JSTL ou podemos usar outras ferramentas como Freemarker ou Velocity.

O VRaptor é um framework MVC que estrutura a forma em que os Controllers vão trabalhar, facilitam o desenvolvimento da camada de Modelo com componentes e injeção de dependências e, na parte da camada de Visualização, o VRaptor nos provê formas de se integrar facilmente com qualquer uma das bibliotecas. Mas o VRaptor deixa livre a escolha das bibliotecas de visualização, o que também significa que ele não vem com nenhuma delas por padrão, além do JSP e JSTL do servidor.

Nesse capítulo, veremos algumas formas de interação com a camada de visualização, usando a biblioteca javascript jQuery, mas não iremos muito a fundo, pois foge do escopo deste livro. Outros livros da Casa do Código ajudam nessa parte, como por exemplo o "Dominando JavaScript com jQuery", escrito pelo Plínio Balduíno.

Importante salientar que você, como desenvolvedor Web, não precisa necessariamente ser um ás de HTML, CSS e Javascript, mas é muito importante conhecer o que é possível fazer e conseguir trabalhar pelo menos com o básico dessas tecnologias para que, mesmo que exista um desenvolvedor front-end na sua equipe, você saiba como fazer a integração com a aplicação no servidor.

O navegador não roda JSPs!

Resolvi colocar esse *disclaimer* no livro, pois é uma confusão bastante comum de desenvolvedores Java que trabalham para a Web, principalmente no começo da carreira. Quando editamos as páginas do sistema, usamos um arquivo JSP, que é o template padrão para criar páginas HTML em Java.

Mas o JSP é apenas uma ferramenta para facilitar a geração dinâmica de HTML do lado do servidor, com o uso de variáveis, `ifs`, `fors` e etc. No entanto, o JSP roda apenas **do lado do servidor**, ou seja, é executado usando as variáveis que passamos na requisição e gera um HTML, que é enviado para o navegador. Ou seja, quando vemos a página no nosso browser, o que estamos vendo é o HTML gerado pelo JSP correspondente, e não mais o JSP!

Isso significa que, uma vez que o HTML foi para o navegador, não temos mais acesso a variáveis nem a estruturas do JSP. Se quisermos executar alguma lógica que modifica os elementos da página, precisamos usar Javascript, que é a linguagem de programação disponível no browser.

Para ficar claro: o JSP é executado do lado do servidor para produzir um HTML. Este HTML é enviado para o navegador que fez a requisição e a página gerada será

mostrada. Para melhorar a usabilidade e interagir com os elementos da página sem sair dela, precisamos usar Javascript, pois o JSP não está mais disponível.

12.1 Executanto uma operação pontual: Remoção de livros

No nosso sistema, temos uma listagem de livros, onde conseguimos ver algumas das informações cadastradas nesses livros. Esta listagem é gerada pelo seguinte código JSP:

```
<h3>Lista de Livros</h3>
<ul>
<c:forEach items="${livroList}" var="livro">
  <li>
    <img
      src="${linkTo[LivrosController].capa[livro.isbn] }"
      width="70" height="100"/>

    ${livro.titulo} - ${livro.descricao} -

    <a href="${linkTo[LivrosController].edita[livro.isbn] }">
      Modificar
    </a>
  </li>
</c:forEach>
</ul>
```

E produz a listagem de livros.

Vamos modificar esta listagem para incluir um link para remoção do livro em questão. Esse link será incluído após o link de modificar o livro:

```
<c:forEach items="${livroList}" var="livro">
  <li>
    ...
    <a href="${linkTo[LivrosController].edita[livro.isbn]}">
      Modificar
    </a>
    -
    <a href="${linkTo[LivrosController].remove[livro.isbn]}">
      Remover
```

```
    </a>
  </li>
</c:forEach>
```

Para isso funcionar, precisamos criar esse novo método no controlador seguindo a interface REST, ou seja, usando o método http DELETE e com a URL de um livro específico:

```
@Resource
public class LivrosController {
  //...

  @Delete("/livro/{isbn}")
  public void remove(String isbn) {
    Livro livro = estante.buscaPorIsbn(isbn);

    estante.retira(livro);
  }
}
```

Esse novo método cria alguns problemas: Primeiro, o método é DELETE mas estamos usando um link, que executa o método GET. Pelo menos deveríamos estar usando o método POST, pois estamos modificando algo no servidor. O segundo problema é que, mesmo que consigamos executar o método `remove`, o que deve acontecer, visualmente? Poderíamos levar para uma página dizendo que o produto foi removido com sucesso ou poderíamos redirecionar de volta para a lista de livros, agora com o livro em questão removido. Neste último caso, estamos carregando a página inteira para apenas apagar um dos itens da lista.

Podemos melhorar essa usabilidade usando Javascript: ao clicar em "Remover", fazemos a requisição ao de remoção do livro e, se der tudo certo, apagamos a linha do livro, sem sair da página! Vamos fazer isso com a ajuda do jQuery, que facilita bastante esse tipo de operações. Como criamos o projeto usando o VRaptor Scaffold, o jQuery já está incluído mas, caso tivéssemos criado o projeto de outra maneira, precisaríamos baixar o jQuery (http://jquery.com/download) e incluí-lo na página.

Antes de poder executar o javascript, precisamos preparar a nossa listagem para facilitar a interação com os elementos. Um dos jeitos de fazer isso é dando ids ou classes para os elementos que serão importantes para a nossa lógica javascript. Ids precisam ser únicos na página, ou seja, só podemos colocá-los em elementos que não

se repetem, como é o caso da lista. Classes podem ser colocadas em vários elementos, portanto vamos usá-las nos itens da lista e no link de remoção:

```
<ul id="livros">
<c:forEach items="${livroList}" var="livro">
  <li class="livro">
    ...
    <a href="${linkTo[LivrosController].edita[livro.isbn] }">
      Modificar
    </a>
    -
    <a class="remove"
       href="${linkTo[LivrosController].remove[livro.isbn] }">
      Remover
    </a>
  </li>
</c:forEach>
</ul>
```

Para adicionar o comportamento ao link de remoção, precisamos colocar código javascript ou dentro de uma tag `<script>` dentro da página, ou em um arquivo `.js` separado que vamos incluir na página. Para ficar mais simples, vamos fazer a primeira opção:

```
<script>
  ...
</script>
```

Este código precisa rodar depois que o browser tiver lido o HTML da lista, então o colocamos dentro do bloco de inicialização do jQuery:

```
<script>
  $(function() {
    //...
  });
</script>
```

Agora precisamos adicionar um comportamento a todos os links de remoção. E esse comportamento será disparado assim que clicarmos no link. Para isso, vamos, dentro da listagem de livros (`#livros`), procurar todos os links de remoção (`.remove`) e executar um código ao clicar:

```
$(function() {
  $("#livros .remove").on("click", function() {

  });
});
```

`#livros` indica o elemento com o id `livros` e `.remove` indica todos os elementos que têm a classe `remove`. Ao fazermos `$("#livros .remove")` estamos representando todos os elementos da classe `remove` que estão dentro do elemento `#livros`. O resto da expressão diz que, ao clicar (`on("click"`), vamos executar a função passada.

Apesar de ser um link, não queremos que o `Remover` saia da página, então precisamos indicar que não queremos esse comportamento padrão, recebendo um parâmetro na função que representa o evento do clique e chamando a função `preventDefault`:

```
$("#livros .remove").on("click", function(event) {
  event.preventDefault();

});
</script>
```

Vamos cuidar primeiro da parte visual da operação: queremos que a linha do livro suma após a remoção. Dentro da função que passamos para tratar o clique, o objeto `this` se refere ao próprio link. Tudo o que precisamos fazer é achar a linha do livro correspondente ao link que foi clicado. Como o link está dentro do `li` do livro, podemos usar o método `closest`, que procura o primeiro pai do elemento que bate com o seletor passado. Para não depender do elemento HTML que usamos, vamos usar a classe `livro`:

```
$("#livros .remove").on("click", function(event) {
  event.preventDefault();

  var livro = $(this).closest(".livro");
});
```

Com o elemento do livro em mãos, podemos escondê-lo usando o método `fadeOut`, que faz com que o elemento suma aos poucos:

```
$("#livros .remove").on("click", function(event) {
  event.preventDefault();
```

```
  var livro = $(this).closest(".livro");

  livro.fadeOut();
});
```

Pronto, agora ao clicar no link, a linha do livro vai sumir. Mas apenas manipulamos o HTML do browser, nada foi enviado ao servidor. Precisamos executar a requisição que remove, de fato, o livro, mas sem sair da página. A técnica que usamos para fazer isso é o que chamamos de AJAX, que originalmente significa *Asynchronous Javascript And XML*, ou seja, uma requisição assíncrona que usa Javascript e XML, mas pode usar outras coisas além de XML para trafegar os dados, como por exemplo JSON.

O código nativo para gerar essa requisição AJAX no browser é um pouco trabalhoso, mas podemos usar o jQuery para facilitar esse trabalho, através da função `$.ajax`. Passamos para essa função um objeto contendo os atributos da requisição, como a URL a ser chamada, o método da requisição e os parâmetros. No nosso caso a URL da requisição é a mesma do link que clicamos, e conseguimos acessá-la através do atributo `href`:

```
$("#livros .remove").on("click", function(event) {
  // ...

  $.ajax({
    url: $(this).attr("href")
  });
});
```

Por padrão, essa requisição usará o método `GET`, mas precisamos do método `DELETE`. O atributo para mudar o método da requisição é o `type` e poderíamos mudá-lo:

```
$("#livros .remove").on("click", function(event) {
  // ...

  $.ajax({
    url: $(this).attr("href"),
    type: 'DELETE'
  });
});
```

Embora isso seja possível, não é suportado por todos os browsers. Para garantir que vai funcionar, podemos trocar o método para `POST` e usar o parâmetro `_method: DELETE`, que podemos passar através do atributo `data`:

```
$("#livros .remove").on("click", function(event) {
  // ...

  $.ajax({
    url: $(this).attr("href"),
    type: 'POST',
    data: { _method: "DELETE" }
  });
});
```

Dessa forma, executamos a requisição AJAX que vai remover o livro do banco de dados. Como essa requisição é assíncrona, o código não fica esperando ela acabar. Além disso, a requisição pode terminar com sucesso ou dar algum erro. Se quisermos tratar cada um desses casos, precisamos continuar a configuração do `$.ajax`.

Para isso, podemos passar funções que serão chamadas assim que a requisição terminar com sucesso, ou com erro, ou simplesmente terminar. Chamamos essa função de **callback**, pois será chamada assim que a resposta da requisição voltar:

```
$.ajax({
  url: $(this).attr("href"),
  type: 'POST',
  data: { _method: "DELETE" }
}).done(function(data, textStatus, jqXHR) {
  // executada em caso de sucesso
}).fail(function(jqXHR, textStatus, errorThrown) {
  // executada em caso de erro
}).always(function() {
  // executada assim que a requisição termina,
  // seja com sucesso ou com erro
});
```

No nosso caso, só queremos remover a linha do livro caso a requisição seja com sucesso. Em caso de erro, podemos apenas mandar uma mensagem de alerta para o usuário. O código completo ficaria:

```
$("#livros .remove").on("click", function(event) {
  event.preventDefault();
```

```
  var livro = $(this).closest(".livro");

  $.ajax({
    url: $(this).attr("href"),
    type: 'POST',
    data: { _method: "DELETE" }
  }).done(function(data, textStatus, jqXHR) {

    livro.fadeOut();

  }).fail(function(jqXHR, textStatus, errorThrown) {

    alert("O Livro não foi removido!");

  });
});
```

Dessa forma, completamos a remoção do livro usando AJAX, pelo menos na parte do browser. Na parte do servidor, tudo o que precisamos é gerar uma resposta com sucesso, caso o livro tenha sido removido. Não precisamos mostrar uma página nem redirecionar a requisição. Para isso, usamos o método `result.nothing()`.

```
@Resource
public class LivrosController {
  //...

  @Delete("/livro/{isbn}")
  public void remove(String isbn) {
    Livro livro = estante.buscaPorIsbn(isbn);

    estante.retira(livro);

    result.nothing();
  }
}
```

Repare que o método do Controller é um método normal, não precisa de nenhuma configuração complicada para responder uma chamada AJAX. O máximo que fizemos foi dizer que não vamos retornar nenhum resultado, além do status de

sucesso. Mas caso retornássemos algum resultado, ele estaria disponível na variável `data` da função passada para o `done` da chamada `ajax`.

Capítulo 13

Apêndice: Containers de Injeção de Dependência

Uma das principais características do VRaptor é que todos os componentes já nascem com injeção de dependências disponível, de modo que conseguimos receber pelo construtor todas as classes das quais um componente ou um controller dependem.

Essa característica é tão importante que os próprios componentes internos do VRaptor são criados e coordenados usando injeção de dependências. No entanto, fazer esse controle é algo bastante complicado e, por isso, o VRaptor delega essa responsabilidade para um container de injeção de dependências, que é quem vai criar e gerenciar todos os componentes do VRaptor e da aplicação.

Existem vários containers disponíveis no mercado, cada um com uma forma diferente de configuração, cada um com seu conjunto de funcionalidades extras. E para dar mais opções para que o VRaptor se adeque melhor à sua aplicação, existe a integração com quatro containers diferentes: o Pico Container, o Spring, o Guice e,

ainda em fase de testes, o CDI.

Para instalar cada um desses containers, só é necessário adicionar os jars do container no classpath, que o VRaptor detecta o container automaticamente. É importante que não existam jars dos outros containers, para que o VRaptor possa escolher o container correto. A exceção dessa regra é o CDI, que tem uma instalação um pouco mais complicada, conforme veremos a seguir.

As classes que fazem a integração com os containers se chamam **Providers** no VRaptor. Neste capítulo, vamos ver as características e as funcionalidades extra de cada container.

13.1 Pico Container

O Pico Container foi um dos primeiros containers de injeção de dependência que surgiram em Java e sua principal característica é ser extremamente leve. Ele é focado apenas na injeção de dependências e faz isso da maneira mais simples possível: registramos implementações para os componentes, e toda a injeção é feita pelo construtor.

Se fôssemos configurá-lo para criar objetos do nosso sistema, essa configuração poderia ser:

```
MutablePicoContainer container = new DefaultPicoContainer();
                    // interface     implementação
container.addComponent(Acervo.class, AcervoNoAdmin.class);
container.addComponent(ClienteHTTP.class, URLClienteHTTP.class);

Acervo acervo = container.getComponent(Acervo.class);
```

Para que a configuração seja simples assim, as dependências precisam ser recebidas no construtor, que também é o padrão do VRaptor.

O Provider de Pico Container (`PicoProvider`) foi o primeiro a ser implementado no VRaptor 3, justamente por ser o mais simples de configurar e já ter um padrão bom a ser seguido. Para conseguirmos implementar os escopos (`request`, `session` e `application`) usamos a funcionalidade de containers filhos (child containers), na qual os filhos conseguem usar as dependências do pai mas o pai não consegue usar as dependências dos filhos.

O pai de todos é o container do `@ApplicationScoped`. Todos os outros escopos conseguem receber como dependência um `@ApplicationScoped`, mas não o

contrário. Um componente no escopo de aplicação pode ser criado fora de uma requisição, portanto não pode receber como dependência um componente de escopo de requisição ou de sessão. Outro motivo é que, mesmo que pudesse, a dependência só valeria para a primeira requisição, após isso estaria destruída e o componente de escopo de aplicação estaria usando como dependência um objeto destruído.

O provider de Pico Container não possui funcionalidades extra, justamente porque ele se foca na injeção de dependência apenas. Por esse motivo, é o provider mais rápido dos implementados no VRaptor.

13.2 Spring IoC

O Spring é um framework criado há mais de 10 anos para facilitar o desenvolvimento de aplicações Java e suprir as várias deficiências que existiam no J2EE (atual Java EE). É uma das bibliotecas mais famosas no mundo Java, principalmente por ser extremamente modularizado e possuir módulos que resolvem os mais variados problemas que enfrentamos ao desenvolver aplicações Java.

O módulo mais importante é o **IoC** (*Inversion of Control*), que cuida da injeção de dependências. É o módulo central do Spring, que coordena todos os componentes dos outros módulos. É também o módulo que o VRaptor usa para implementar o provider de injeção de dependências que usa o Spring.

Assim como o Pico ou o Guice, usar o Spring como provider é totalmente transparente, não muda nada do uso do VRaptor, da criação dos componentes ou da injeção pelo construtor. No entanto, por rodar dentro do Spring, temos a possibilidade de usar qualquer um dos módulos do Spring integrados automaticamente com os componentes criados pelo VRaptor, e vice-versa. Basta configurar os módulos e componentes no Spring e eles já ficam disponíveis para injeção de dependências nos componentes do VRaptor.

Podemos usar algumas das anotações do Spring automaticamente, sem a necessidade de configurações adicionais. É o caso do `@Autowired`, que habilita a injeção via setter, atributo ou init method:

```
@Component
public class MeuComponenteNoVRaptor {

  @Autowired
  private Estante estante;
```

```
  @Autowired
  public void setAcervo(Acervo acervo) {...}

  @Autowired
  public void inicializa(Result result, Validator validator) {...}

}
```

O Spring foi criado numa época em que o Java era apenas uma ferramenta para transformar grandes quantidades de XML em stack traces, portanto seu meio principal de configuração é um XML onde definimos o contexto da aplicação. Para que essa configuração se integre automaticamente com o VRaptor, precisamos criá-la num arquivo chamado `applicationContext.xml` no classpath (ex. pasta `src/main/resources`).

Uma boa configuração que podemos colocar nesse xml é a que habilita a criação de componentes via anotações, aumentando bastante a produtividade diminuindo consideravelmente a necessidade de editar o arquivo de configurações:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd"
>

  <context:component-scan base-package="br.com.casadocodigo.livraria"/>

</beans>
```

Assim, podemos criar e configurar componentes usando as anotações do Spring, por exemplo, usando `@Repository` ou `@Service`. Mas, ao criar componentes usando o Spring, você está sujeito aos seus padrões: nenhuma injeção é automática e, mesmo que seja pelo construtor, é preciso anotar a dependência com `@Autowired`. O escopo padrão é o "singleton", equivalente ao `@ApplicationScoped` do VRaptor, então se quisermos mudar o escopo usamos a anotação `@Scope`.

Para exemplificar a integração com um dos módulos do Spring, vamos usar o **Spring Transaction**, que habilita o uso da anotação `@Transactional` para controlar as transações nos métodos dos componentes. Assim, se eu estou num DAO e quero uma transação em volta do método `salva`, faríamos:

```
@Component
public class JPALivroDAO implements LivroDAO {

  @Transactional
  public void salva(Livro livro) {
    manager.persist(livro);
  }
}
```

Se formos executar mais de uma operação no banco de dados, na classe `EstanteNoBancoDeDados` que usa o `LivroDAO`, poderíamos usar novamente a anotação `@Transactional`:

```
@Component
public class EstanteNoBancoDeDados implements Estante {

  @Transactional
  public void guarda(Livro livro) {
    livroDao.adiciona(livro);
    estoqueDao.entrada(livro);
  }

}
```

Nesse caso, o Spring sabe que não precisa abrir uma transação dentro de outra e apenas abre a transação mais externa. Para que tudo isso funcione, precisamos indicar no `applicationContext.xml` que vamos usar o spring transactional e que ele vai ser configurado através de anotações:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  ...
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="...
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

```
  ...

  <tx:annotation-driven />

</beans>
```

Existem várias maneiras possíveis de se abrir e commitar uma transação com o banco de dados, cada uma dependendo da tecnologia que usamos para se comunicar com ele. No nosso caso, precisamos falar para o Spring gerenciar as transações usando a JPA. E já que o Spring vai controlar as transações, precisamos que ele também controle a abertura e o fechamento dos `EntityManagers`.

```
<tx:annotation-driven />

<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="emFactory" />
</bean>

<bean id="emFactory"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="default"/>
</bean>
```

Dessa forma, precisamos mudar a injeção do `EntityManager` para ser anotada com `@PersistenceContext`. Nesse caso, vamos mudar a injeção por atributo. Além disso, para termos acesso ao `EntityManager` para fazer consultas ao banco de dados, precisamos garantir que ele estará disponível, usando o `@Transactional`. Como para consultas não precisamos modificar o banco, usamos o atributo `readonly`:

```
@Transactional(readonly=true)
@Component
public class JPALivroDAO implements LivroDAO {

  @PersistenceContext
  private EntityManager manager;

  @Transactional // para que ele não seja readonly nesse caso
  @Override
```

```
  public void adiciona(Livro livro) {
    if (livro.getId() == null) {
      this.manager.persist(livro);
    } else {
      this.manager.merge(livro);
    }
  }

  @Override
  public List<Livro> todos() {
    return this.manager
        .createQuery("select l from Livro l", Livro.class)
        .getResultList();
  }

  @Override
  public Livro buscaPorIsbn(String isbn) {
    try {
      return this.manager
          .createQuery("select l from Livro l "
          + " where l.isbn = :isbn", Livro.class)
          .setParameter("isbn", isbn)
          .getSingleResult();
    } catch (NoResultException e) {
      return null;
    }
  }

}
```

Podemos usar esse e qualquer outro módulo do Spring, em conjunto com os componentes criados pelo VRaptor, pois, ao usá-lo com o Spring como container, todos os componentes do VRaptor e da sua aplicação são registrados como beans do Spring. A única diferença para um bean normal do Spring são os escopos gerenciados pelas anotações do VRaptor e a injeção por construtor por padrão.

A desvantagem em usar o Spring é que, por conter tantos módulos e funcionalidades, a injeção de dependências é bem mais lenta que no Guice ou no Pico, portanto se não for usar nenhuma das funcionalidades adicionais do Spring, o Guice e o Pico são opções mais rápidas.

13.3 Google Guice

O Guice (pronunciado como 'juice') é uma biblioteca de injeção de dependências disponibilizada pelo Google em https://code.google.com/p/google-guice/. O Guice se aproveita do sistema de tipos do java para tornar sua configuração e o seu uso mais simples. Desde a versão 3.4 do VRaptor é o provider padrão do VRaptor.

A configuração do Guice se dá por módulos, onde registramos quais são as implementações de cada uma das interfaces que usaremos no sistema. Para a nossa livraria, seria algo do tipo:

```
public class LivrariaModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Estante.class).to(EstanteNoBancoDeDados.class);
    bind(LivroDAO.class).to(JPALivroDAO.class);
    bind(LivrosController.class);
    //...
  }
}
```

Ao usar essa configuração, o Guice obriga que os pontos de injeção sejam anotados com `@Inject`. Portanto, deveríamos anotar todos os construtores com `@Inject`. Para poder conseguir uma instância criada e gerenciada pelo Guice, criamos um injector:

```
Injector injector = Guice.createInjector(new LivrariaModule());

LivrosController controller =
    injector.getInstance(LivrosController.class);
```

O provider do Guice para o VRaptor, no entanto, normaliza a injeção do Guice para usar os padrões do VRaptor, como os escopos e a injeção automática pelo construtor. Mas podemos usar qualquer outro recurso do Guice, assim como fizemos com o Spring, já que o VRaptor roda dentro do ambiente de injeção de dependências do Guice.

Para instalar uma funcionalidade adicional do Guice, precisamos estender o `GuiceProvider` do VRaptor e configurar o modulo adicional:

```
public class CustomProvider extends GuiceProvider {
```

```
  @Override
  protected Module customModule() {
    return new Module {
      public void configure(Binder binder) {
        // para manter o comportamento padrão do VRaptor
        binder.install(super.customModule());

        binder.install(new OutroModulo());

        //...
      }
    };
  }

}
```

Para que o VRaptor use essa configuração, precisamos registrar no web.xml:

```
<context-param>
  <param-name>br.com.caelum.vraptor.provider</param-name>
  <param-value>br.com.pacote.do.CustomProvider</param-value>
</context-param>
```

Mesmo possuindo funcionalidades adicionais, que podem ser encontradas em https://code.google.com/p/google-guice/wiki/Motivation, o Guice possui uma performance muito parecida com a do Pico, sendo uma opção de provider bastante vantajosa.

13.4 CDI

O CDI é a especificação do Java que diz respeito à Injeção de Dependências e foi introduzida no java para servidores no Java EE 6 e possibilita o acesso de qualquer classe a todos os recursos dos servidores de aplicação que antes só eram acessíveis a EJBs.

Em classes gerenciadas pelo CDI, podemos acessar EJBs do servidor, usando a anotação `@EJB`, receber um `EntityManager` configurado, usando a anotação `@PersistenceContext`, acessar algum recurso do servidor como o agendador de tarefas `TimerService`, com a anotação `@Resource`, e assim por diante.

Podemos, ainda, usar o CDI em modo standalone, ou seja, fora do servidor de

aplicação, com a ajuda do Weld, sua implementação de referência. Assim, podemos usá-lo em aplicações web normais, instaladas no Tomcat ou no Jetty.

O provider do CDI para VRaptor ainda não foi oficialmente lançado, pois faltam alguns últimos detalhes para facilitar o seu uso em aplicações feitas com o VRaptor. Isso porque ele obriga que todas as classes tenham um construtor sem argumentos, que todo projeto tenha um arquivo chamado `beans.xml` na pasta `META-INF` e algumas outras exigências.

Mais informações sobre esse provider estão em: https://github.com/caelum/vraptor-cdi-provider

Escolha o provider que mais se adequa ao seu projeto e, caso vá apenas usar a injeção de dependências do VRaptor, todas as implementações mantêm o mesmo comportamento caso você use as anotações do VRaptor.

Capítulo 14

Apêndice: Plugins para o VRaptor

O VRaptor é um framework Web MVC que se concentra bastante na parte do controller, ou seja, em tratar requisições e dispará-las para algum dos Controllers da sua aplicação. No entanto, o VRaptor roda dentro de um container de injeção de dependências, onde você consegue criar diversos componentes que interagem entre si para conseguir executar toda a lógica da sua aplicação.

Essa arquitetura de componentes facilita bastante o reúso de componentes, então conseguimos usar as mesmas soluções de segurança, transações, controle de ambientes, relatórios entre diversas aplicações diferentes.

E para facilitar o compartilhamento desses componentes, o VRaptor possui uma funcionalidade para criar e usar plugins, que são conjuntos de classes que resolvem algum problema e podem ser adicionados facilmente em qualquer aplicação. Uma vez criado o plugin, basta adicionar o seu jar ao classpath e todos os seus componentes podem ser usados.

Foi criado, também, um catálogo de todos os plugins do VRaptor, alguns que nasceram de projetos da própria Caelum e outros feitos pela comunidade de usuá-

rios. Este catálogo que está disponível no Github em https://github.com/caelum/vraptor-contrib contendo referências aos respectivos projetos dos plugins no Github.

Nesse capítulo, vamos ver alguns desses plugins, os oficiais e os mais utilizados, mas a partir do catálogo podemos ver a documentação de cada plugin para obter as suas informações de uso.

14.1 VRaptor JPA

É o plugin responsável por criar e gerenciar o `EntityManager` para que possamos executar as operações no banco de dados usando a JPA. O código fonte deste plugin pode ser encontrado em https://github.com/caelum/vraptor-jpa.

Este plugin estava dentro do jar do VRaptor até a versão 3.4, mas foi extraído na versão 3.5 para tornar o jar do VRaptor mais simples e desacoplado do plugin. Para usá-lo precisamos do seu jar, que está no repositório do Maven e podemos usá-lo declarando a sua dependência e alguma das implementações da JPA no pom.xml:

```
<dependency>
    <groupId>br.com.caelum.vraptor</groupId>
    <artifactId>vraptor-jpa</artifactId>
    <version>1.0.0</version>
</dependency>

<!-- Hibernate, OpenJPA ou alguma outra implementação da JPA -->
```

Ou baixe o jar direto do repositório do maven:

http://repo1.maven.org/maven2/br/com/caelum/vraptor/vraptor-jpa/

Com esse plugin podemos simplesmente receber o `EntityManager` no construtor dos componentes que se comunicarão com o banco de dados:

```
@Component
public class LivroDAO {

  private EntityManager manager;

  public LivroDAO(EntityManager manager) {
    this.manager = manager;
  }

  public void salva(Livro livro) {
```

```
    this.manager.persist(livro);
  }
}
```

Este EntityManager será aberto ao começo da requisição e fechado ao final dela. O plugin também abre uma transação ao começo da requisição e automaticamente faz o commit se tudo der certo ou o rollback caso exista algum erro de validação ou alguma exception lançada.

Para usar este plugin também é necessário haver um persistence.xml com uma persistence unit chamada default:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
    <!-- Dados de conexão com o banco de dados -->
  </persistence-unit>

</persistence>
```

Como o EntityManager vive durante a requisição, só pode ser usado em componentes que são @RequestScoped (o escopo padrão), ou @PrototypeScoped que são usados em componentes de escopo de requisição. Caso seja necessário usar um EntityManager em componentes de outros escopos, receba um EntityManagerFactory no construtor e controle a abertura e o fechamento do EntityManager manualmente, de acordo com as necessidades do componente.

14.2 VRaptor Hibernate e VRaptor Hibernate 4

Assim como o plugin **VRaptor JPA**, este plugin estava dentro do jar do VRaptor e foi extraído para um plugin à parte na versão 3.5 do VRaptor. É o plugin responsável por criar e gerenciar a Session do Hibernate para acessarmos o banco de dados. Como o Hibernate mudou bastante a sua API da versão 3 para a versão 4, temos dois plugins diferentes, um para cada versão:

- VRaptor Hibernate: https://github.com/caelum/vraptor-hibernate para a versão 3 do Hibernate. Para usá-lo adicione a seguinte dependência no pom.xml:

  ```
  <dependency>
      <groupId>br.com.caelum.vraptor</groupId>
      <artifactId>vraptor-hibernate</artifactId>
      <version>1.0.0</version>
  </dependency>
  ```

  Ou baixe-o diretamente do maven:

  http://central.maven.org/maven2/br/com/caelum/vraptor/vraptor-hibernate/

- VRaptor Hibernate 4: https://github.com/garcia-jj/vraptor-plugin-hibernate4 para a versão 4 do Hibernate. Para usá-lo adicione a seguinte dependência no pom.xml:

  ```
  <dependency>
      <groupId>br.com.caelum.vraptor</groupId>
      <artifactId>vraptor-plugin-hibernate4</artifactId>
      <version>1.0.2</version>
  </dependency>
  ```

  Ou baixe-o diretamente do maven:

  http://central.maven.org/maven2/br/com/caelum/vraptor/vraptor-plugin-hibernate4/

O funcionamento desses dois plugins é idêntico e bem parecido com o plugin da JPA: Uma `Session` é aberta ao início da requisição e fechada ao final. Também é aberta uma transação ao início da requisição e feito o commit ou rollback dela ao final, assim como no plugin da JPA. Para usá-lo, basta receber a `Session` no construtor do componente desejado:

```
@Component
public class LivroDAO {

  private Session session;

  public LivroDAO(Session session) {
    this.session = session;
```

```
  }

  public void salva(Livro livro) {
    this.session.save(livro);
  }
}
```

Também é necessário ter o `hibernate.cfg.xml` ou o `hibernate.properties` no classpath, para configurar as credenciais do banco e outros parâmetros do Hibernate.

14.3 VRaptor Environment

Praticamente toda aplicação que desenvolvemos é executada em pelo menos dois ambientes diferentes: a máquina do desenvolvedor e o servidor de produção. Isso quando não existem outros ambientes como testes, QA, homologação, pré-produção, entre outros. A grande maioria do código da aplicação deveria rodar do mesmo jeito independente do ambiente que estamos usando.

No entanto, algumas configurações **precisam** ser diferentes em cada ambiente. Talvez o mais óbvio seja o banco de dados: não queremos usar o mesmo banco de produção enquanto estamos desenvolvendo a aplicação. Além disso, precisam ser diferentes as credenciais usadas para enviar e-mails, o endereço de algum serviço que a aplicação usa, presença ou não de cache, entre outros.

Para facilitar essa mudança de ambientes podemos usar o plugin **VRaptor Environment**, com o código fonte e documentação em: https://github.com/caelum/vraptor-environment

Para instalá-lo, assim como os outros, colocamos a sua dependência no pom.xml:

```
<dependency>
    <groupId>br.com.caelum.vraptor</groupId>
    <artifactId>vraptor-environment</artifactId>
    <version>1.1.4</version>
</dependency>
```

Ou baixando o jar em:

http://central.maven.org/maven2/br/com/caelum/vraptor/vraptor-environment

Com esse plugin no classpath, precisamos informar em qual ambiente estamos, com o seguinte parâmetro no web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.environment</param-name>
    <param-value>development</param-value>
</context-param>
```

Nesse caso, configuramos o ambiente `development`, ou seja, o ambiente de desenvolvimento. Assim, para mudar o ambiente, editamos o web.xml, ou melhor, sobrescrevemos esse parâmetro diretamente no servidor de produção. Cada servidor tem uma maneira de sobrescrever um context-param da aplicação. Por exemplo, no Tomcat e no Jetty:

http://tomcat.apache.org/tomcat-6.0-doc/config/context.html#Context_Parameters

http://wiki.eclipse.org/Jetty/Reference/override-web.xml

Com isso podemos criar um arquivo properties no classpath (pasta `src` ou `src/main/resources` em geral) com o nome do ambiente, contendo todas as configurações deste ambiente. No nosso exemplo, criaríamos o `development.properties`, com algumas configurações:

```
email.suporte = dev@livraria.com
servidor.smtp = localhost:25

url.admin = http://localhost:8080/livraria-admin
```

E para usar a configuração do ambiente, recebemos no construtor a classe `Environment`, que possui o método `get` que acessa uma das configurações:

```
@Component
public class AcervoNoAdmin implements Acervo {

  private ClienteHTTP http;
  private String urlAdmin;

  public AcervoNoAdmin(ClienteHTTP http, Environment env) {
    this.http = http;
    this.urlAdmin = env.get("url.admin");
  }

  @Override
  public List<Livro> todosOsLivros() {
    String xml = http.get(urlAdmin + "/integracao/listaLivros");
```

```
    //...
  }
}
```

Também é possível resgatar algum arquivo de configuração que esteja numa pasta com o nome do ambiente dentro do classpath. No nosso caso, se quisermos acessar o arquivo `admin.cfg.xml` de desenvolvimento, que deve estar em `src/main/resources/development/admin.cfg.xml` podemos usar:

```
URL cfg = env.getResource("admin.cfg.xml");
```

Este plugin ainda se integra com outros plugins como o **VRaptor Simple Mail** e o **VRaptor Hibernate**, já que quase sempre precisamos de configurações específicas por ambiente para eles.

14.4 VRaptor Simple Mail e VRaptor Freemarker

Uma tarefa bastante comum a aplicações Web é enviar e-mails aos seus usuários, seja para avisar que uma conta foi criada, enviar dados de uma compra, promoções, notificações etc. Para que isso seja possível, precisamos de um servidor SMTP para o envio de e-mail, com credenciais válidas, um endereço de e-mail para usarmos como origem (campo `From`).

O plugin **VRaptor Simple Mail** tem como objetivo facilitar a criação e o envio de e-mails na aplicação. O seu código fonte e uma ótima documentação estão em https://github.com/caelum/vraptor-simplemail e pode ser instalado usando a dependência:

```
<dependency>
  <groupId>br.com.caelum.vraptor</groupId>
  <artifactId>vraptor-simplemail</artifactId>
  <version>1.2.1</version>
</dependency>
```

Para completar a configuração do plugin precisamos acrescentar ao arquivo de configuração do ambiente (por exemplo o `development.properties`) as seguintes chaves:

```
vraptor.simplemail.main.server = localhost
vraptor.simplemail.main.port = 25
vraptor.simplemail.main.tls = false
```

```
vraptor.simplemail.main.from = no-reply@livraria.com

#se for necessário fazer autenticação:
vraptor.simplemail.main.username = meu-usuario
vraptor.simplemail.main.password = minha-senha
```

Agora, para enviar um e-mail usamos o componente `Mailer` e usamos a API `javax.mail` para criá-lo:

```
@Resource
public class ComprasController {
  private Mailer mailer;

  public ComprasController(Mailer mailer) {
    this.mailer = mailer;
  }

  @Post
  public void finaliza(Compra compra) {
      //...

      Email email = new SimpleMail();
      email.setSubject("Compra efetuada com sucesso!");
      email.addTo(compra.getUsuario().getEmail());
      email.setMsg("Seu pedido de Compra no valor de " +
        compra.getValor() + " foi finalizado com sucesso!");
      mailer.send(email);
    }
}
```

É possível ainda enviar um e-mail de forma assíncrona usando o `AsyncMailer`, assim a requisição não fica parada esperando o e-mail ser enviado.

Porém, se quisermos enviar um e-mail um pouco mais elaborado, com html e imagens, não é muito prático criá-lo dentro de um código Java. Por esse motivo, o Simple Mail se integra com o plugin **VRaptor Freemarker**, no qual podemos usar templates do Freemarker para criar o corpo do e-mail. Para isso, recebemos o componente `TemplateMailer`:

```
@Resource
public class ComprasController {
  private Mailer mailer;
```

```
  private TemplateMailer templates;

  public ComprasController(Mailer mailer, TemplateMailer templates) {
    this.mailer = mailer;
    this.templates = templates;
  }

  @Post
  public void finaliza(Compra compra) {
      //...

      Email email = this.templates
              .template("compraFinalizada.ftl")
              .with("compra", compra)
              .to(compra.getCliente().getNome(),
                  compra.getCliente().getEmail());

      mailer.send(email);
    }
}
```

Para mais informações sobre o VRaptor Freemarker: https://github.com/caelum/vraptor-freemarker

14.5 Agendamento de tarefas: VRaptor Tasks

Quando estamos desenvolvendo aplicações Web, a maioria do trabalho é feito dentro de requisições HTTP. Mas algumas vezes precisamos executar tarefas na aplicação que não serão disparadas por um usuário interagindo com o sistema no navegador, e sim uma tarefa que precisa ser disparada pela própria aplicação.

Isso acontece quando precisamos enviar um e-mail para o usuário 2 horas depois da compra ou quando precisamos consumir dados de um serviço a cada 5 minutos, por exemplo. Para executar esse tipo de tarefa agendada ou periódica, temos uma biblioteca bastante famosa em Java: o Quartz, com a qual podemos criar `Tasks`, que podemos agendar para serem executadas.

Para podermos aproveitar todo o poder do Quartz junto com a injeção de dependências que o VRaptor nos dá, foi criado o plugin **VRaptor Tasks** pelo William Pivotto, e pode ser encontrado em https://github.com/wpivotto/vraptor-tasks. Esse plugin tem uma ótima documentação na página do github, então vou apenas dar

alguns exemplos do que é possível fazer com ele.

Para criar tarefas periódicas, que não dependem de componentes `@RequestScope`, devemos criar uma `Task` e anotá-la com `@Scheduled`, configurando a frequência.

```
@ApplicationScoped //ou @PrototypeScoped
@Scheduled(cron="0 */5 * * * ?")
public class Sincronizador implements Task {
  public Sincronizador(ClienteHTTP http) {...}

  public void execute() {
    // sincroniza com um serviço qualquer...
  }
}
```

Caso a tarefa precise de componentes de escopo de requisição, criamos um controller e anotamos os métodos que queremos agendar com `@Scheduled`:

```
@Resource
public EnviadorDeEmailController {
  public EnviadorDeEmailController(
    UsuarioRepository usuarios,
    Mailer mailer) {
    //...
  }

  @Scheduled(cron="0 0 * * * ?")
  public void avisaSobreCadastro() {
    List<Usuario> avisaveis = usuarios.queNaoPreencheramCadastro();
    // manda email pra todos
  }
}
```

Nesse caso, todo dia à meia-noite uma tarefa executará uma requisição que cairá nesse método, assim todos os componentes `@RequestScoped` estarão disponíveis. Ainda é possível agendar tarefas em tempo de execução, gerenciar e monitorar as tarefas já agendadas.

14.6 Controle de usuários: VRaptor SACI

Na seção 9.3 vimos como criar uma infraestrutura de controle de usuários no sistema, com um novo modelo `Usuario` e dois interceptors: o `AutenticacaoInterceptor` para verificar se o usuário está logado ou não e o `AutorizacaoInterceptor` para ver se o usuário tem permissão de acessar o recurso desejado.

No entanto, esse controle de usuários pode ser bastante complicado, por exemplo, se quisermos criar papéis (Roles), ou definir uma hierarquia de usuários, segundo a qual cada um tem permissão de acessar um pedaço das funcionalidades do sistema.

Para facilitar esse controle, o **Diego Maia (bronx)** criou um plugin chamado **VRaptor SACI**, que pode ser encontrado em: https://github.com/bronx/SACI-VRaptor e tem uma ótima documentação de suas funcionalidades.

Para usar esse plugin, precisamos ter uma forma de identificar o usuário logado. Podemos usar um usuário no banco de dados, como fizemos na seção 9.3, ou usar algo mais complexo como LDAP ou AD para isso.

Tínhamos criado a seguinte classe:

```
@Component
@SessionScoped
public class UsuarioLogado {

  private Usuario usuario;

  public void loga(Usuario usuario) {
    this.usuario = usuario;
  }

  public boolean isLogado() {
    return this.usuario != null;
  }

  public Usuario getUsuario() {
    return usuario;
  }

  public void desloga() {
    this.usuario = null;
```

```
  }
}
```

Para adaptar essa classe ao VRaptor SACI, precisamos que ela implemente a interface `Profile`, que define três métodos: `isLoggedIn`, que deve retornar `true` se o usuário está logado, `getRoles`, que deve retornar os papéis ou as permissões do usuário, e `getAccessLevel` que define o nível de acesso do usuário, para permissões hierárquicas.

Adaptando a classe `UsuarioLogado` para essa interface, vamos renomear o método `isLogado` para `isLoggedIn` e implementar os outros métodos de acordo com o nosso modelo:

```
@Component
@SessionScoped
public class UsuarioLogado implements Profile {

  private Usuario usuario;

  public void loga(Usuario usuario) {
    this.usuario = usuario;
  }

  public Usuario getUsuario() {
    return usuario;
  }

  public void desloga() {
    this.usuario = null;
  }

  @Override
  public boolean isLoggedIn() {
    return this.usuario != null;
  }

  @Override
  public List<String> getRoles() {
    // Só temos dois tipos de usuário: admin e não admin:
    if (usuario.isAdmin()) {
      return Arrays.asList("admin");
    } else {
```

```
      return Collections.emptyList();
    }
  }

  @Override
  public int getAccessLevel() {
    // Não usamos permissões hierárquicas, então vamos retornar 0
    return 0;
  }
}
```

Feito isso, não precisamos mais dos interceptors que havíamos criado, pois o SACI já faz a autenticação e a autorização. Mas para ele funcionar corretamente, precisamos indicar qual é a página de login, para ele redirecionar caso o usuário não esteja logado. No nosso caso, a página de login está no método `formulario` do `LoginController`, que precisa ser anotado com `@LoginPage`:

```
@Resource
public class LoginController {
  //...

  @LoginPage
  @Get("/login")
  public void formulario() {}

}
```

Agora o que precisamos fazer é indicar para o SACI quais métodos precisam ser autenticados. Existem duas maneiras de fazer isso. A primeira é quando apenas precisamos que o usuário esteja logado para acessar o controller. Para isso, usamos a anotação `@LoggedIn` num método do controller ou na própria classe do controller, se quisermos que todos os métodos sejam autenticados.

A segunda maneira é definindo o controle de acesso ao método ou ao controller, usando uma ou mais roles (com a anotação `@Role`) ou o nível de acesso (com a anotação `@AccessLevel`). No nosso caso, queremos que o `LivrosController` seja autenticado totalmente, mas apenas alguns métodos precisam ser acessados só pelo admin, então a configuração seria:

```
@LoggedIn
@Resource
```

```
public class LivrosController {

  @Get("/livros/formulario")
  public void formulario() { ... }

  @Get("/livros")
  public List<Livro> lista() { ... }

  @Role("admin")
  @Post("/livros")
  public void salva(Livro livro) { ... }

  //...
}
```

14.7 Criando o seu próprio plugin

Muitas vezes desenvolvemos uma solução legal que precisa ser reutilizada entre diversos projetos da sua empresa, ou simplesmente queremos compartilhar essa solução com outros desenvolvedores. Para que o VRaptor considere essa solução como um plugin, precisamos agrupar todos os seus componentes em um projeto separado, para gerarmos um jar.

Para que esse jar seja registrado automaticamente no VRaptor, ele precisa ter dentro dele o arquivo `META-INF/br.com.caelum.vraptor.packages` com o pacote base do plugin dentro desse arquivo. Vamos supor que queremos transformar o nosso serviço de arquivos que desenvolvemos na seção 8 em um plugin.

Esse serviço é composto de duas classes básicas, o `Diretorio` e o `Arquivo`, e uma implementação que salva os arquivos no banco de dados, o `DiretorioNoBD`. Então vamos extraí-las para um projeto à parte, usando um pacote mais genérico. Esse pacote precisa ser colocado dentro do arquivo `META-INF/br.com.caelum.vraptor.packages` dentro de `src/main/resources`.

```
vraptor-arquivos/
  - src/main/java
    - br.com.casadocodigo.arquivos
      - Arquivo.java
      - Diretorio.java
      - DiretorioNoBD.java
```

```
- src/main/resources
  - META-INF
    - br.com.caelum.vraptor.packages
```

E dentro do arquivo chamado `br.com.caelum.vraptor.packages` colocamos:

```
br.com.casadocodigo.arquivos
```

Agora basta gerar um jar com todos esses arquivos e temos um plugin! Se colocarmos o `vraptor-arquivos.jar` em qualquer projeto com o VRaptor, o `DiretorioNoBD` já será registrado como componente e poderemos receber um `Diretorio` no construtor de qualquer classe da aplicação.

Se o objetivo for compartilhar esse plugin:

- crie um projeto no github, por exemplo em https://github.com/seu-usuario/seu-plugin ;
- suba os arquivos do projeto para lá usando comandos de git em linha de comando o próprio programa do github;
- escreva um arquivo README.md explicando como usar o seu plugin;
- Entre em https://github.com/caelum/vraptor-contrib e siga as instruções para incluir seu plugin na lista;
- Divulgue o seu plugin na lista de emails do VRaptor caelum-vraptor@googlegroups.com ;
- Seja reconhecido por todos =)

Capítulo 15

Apêndice: Funcionamento interno do VRaptor

O VRaptor roda inteiramente dentro do container de injeção de dependências. Isso significa que todo componente do VRaptor é uma classe gerenciada e pode ser substituida pela aplicação, caso ela implemente um componente com a mesma interface.

Neste capítulo veremos como funciona o VRaptor, desde a inicialização do servidor até o tratamento de requisições, mostrando os componentes responsáveis por algumas das funcionalidades e como poderíamos sobrescrever tais componentes.

15.1 Inicialização do Servidor

O ponto de entrada do VRaptor é o filtro com o mesmo nome, que registramos no web.xml (ou é registrado automaticamente se você usa Servlet 3):

```
<filter>
  <filter-name>vraptor</filter-name>
```

```
  <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
</filter>
```

Todo o código da inicialização do VRaptor é iniciado pelo método `init` do filtro. Como tudo roda dentro de um container de injeção de dependências, a primeira coisa feita é a decisão de qual vai ser o provider usado:

```
BasicConfiguration config = new BasicConfiguration(servletContext);
init(config.getProvider());
```

Para essa escolha, a preferência é pelo parâmetro do web.xml chamado `br.com.caelum.vraptor.provider`, que deve apontar para a classe que implementa toda a lógica de injeção de dependências. O uso mais comum desse parâmetro é quando queremos personalizar um pequeno pedaço, como no `GuiceProvider`, para instalarmos um módulo:

```
package br.com.minhaapp.infra;

public class CustomProvider extends GuiceProvider {

  @Override
  protected Module customModule() {
    return new Module {
      public void configure(Binder binder) {
        // para manter o comportamento padrão do VRaptor
        binder.install(super.customModule());

        binder.install(new OutroModulo());

        //...
      }
    };
  }

}
```

Para que o VRaptor use essa configuração, precisamos registrar no web.xml:

```
<context-param>
  <param-name>br.com.caelum.vraptor.provider</param-name>
  <param-value>br.com.minhaapp.infra.CustomProvider</param-value>
</context-param>
```

Caso esse parâmetro não exista, procuramos por classes dos providers no classpath, que só deveriam existir se o jar do provider estiver no projeto. Por esse motivo, não é necessária nenhuma configuração adicional para usar um provider específico, somente o jar no classpath basta. No entanto, existe uma ordem para testar os providers: Spring, Guice e Pico. Se o projeto tiver mais de um dos providers no classpath, essa ordem fará o desempate.

O próximo passo é a inicialização do provider, que envolve o registro de todos os componentes do VRaptor e, em seguida, de todos os componentes da aplicação. O registro dos componentes do VRaptor é feito usando a classe `BaseComponents`, que contém um mapa gigante que ligam as interfaces dos componentes do VRaptor às suas implementações padrão, no escopo correto:

```
public class BaseComponents {

  private final static Map<Class<?>, Class<?>> APPLICATION_COMPONENTS =
    classMap(
        EncodingHandlerFactory.class, EncodingHandlerFactory.class,
        AcceptHeaderToFormat.class, DefaultAcceptHeaderToFormat.class,
        //...
    );

   private static final Map<Class<?>, Class<?>> REQUEST_COMPONENTS =
      classMap(
            MethodInfo.class,   DefaultMethodInfo.class,
            LogicResult.class,  DefaultLogicResult.class,
            PageResult.class,   DefaultPageResult.class,
        //...
      );
  //...
}
```

Já o registro dos componentes da aplicação é feito a partir de um scan de todas as classes que estão em `WEB-INF/classes` e contém anotações do VRaptor, como `@Component`, `@Resource` e `@Intercepts`. O scan não é feito em todas as classes do classpath porque isso é bastante ineficiente, já que teríamos que passar por todas as classes de todos os jars que estão no classpath e que, na maior parte dos casos, não têm nada a ver com o VRaptor.

Além disso, o VRaptor passa por todas as classes que estão abaixo dos pacotes registrados no parâmetro `br.com.caelum.vraptor.packages` do web.xml.

Nesse caso a procura é por todo o classpath, e não somente em WEB-INF/classes. Se algum algum componente a ser registrado estiver dentro de um jar, ele só será escaneado se o seu pacote base estiver nesse parâmetro de packages. Outra maneira de registrar um componente que está dentro de um jar é colocar um arquivo META-INF/br.com.caelum.vraptor.packages contendo o pacote base das classes desse jar, como é feito nos plugins do VRaptor.

Esse scan é feito, por padrão, dinamicamente na inicialização do servidor, usando a biblioteca Scannotation. No entanto, em algumas versões do JBoss que usam um sistema de arquivos virtual para gerenciar as aplicações deployadas, esse scan não funciona corretamente. Nesse caso temos que usar o scan estático, que gera uma classe que contém o código de registro de todos os componentes da aplicação. Esta classe é gerada invocando o método main da classe VRaptorStaticScanning. Isso deve ser feito durante o processo de build da aplicação, pois é gerado um .class na pasta WEB-INF/classes que é lido pelo VRaptor na inicialização do servidor.

Exemplo de como fazer isso usando o Ant, no build.xml:

```
<path id="build.classpath">
    <fileset dir="${webapp.dir}/WEB-INF/lib" includes="*.jar" />
</path>
<target name="vraptor-scanning" depends="compile">
    <java classpathref="build.classpath"
            classname="br.com.caelum.vraptor.scan.VRaptorStaticScanning"
            fork="true">
        <arg value="${webapp.dir}/WEB-INF/web.xml" />
        <classpath refid="build.classpath" />
        <classpath path="${webapp.dir}/WEB-INF/classes" />
    </java>
</target>
```

Exemplo de como fazer isso usando o Maven, no pom.xml:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2.1</version>
  <configuration>
    <mainClass>
      br.com.caelum.vraptor.scan.VRaptorStaticScanning
    </mainClass>
```

```
    <arguments>
      <argument>${basedir}/src/main/webapp/WEB-INF/web.xml</argument>
    </arguments>
  </configuration>
  <executions>
   <execution>
       <phase>process-classes</phase>
       <goals>
           <goal>java</goal>
       </goals>
   </execution>
  </executions>
</plugin>
```

O código de inicialização dos providers, após decidir quais são as classes que devem ser registradas, é uma das partes mais complicadas do funcionamento interno do VRaptor, pois precisamos normalizar o comportamento dos providers para usar as convenções do VRaptor (como escopo de request por padrão e injeção pelo construtor). Além disso é necessário implementar ou configurar os escopos corretamentes e permitir que classes da aplicação (ou de plugins) sobrescrevam componentes internos do VRaptor.

Por ser bastante complexo, vou falar aqui das características principais de cada um dos containers:

- **PicoProvider**: é criado um container (`MutablePicoContainer`) para controlar o escopo de aplicação e todos os outros escopos são criados como containers filhos. A inicialização e finalização de cada escopo é feita pelo próprio VRaptor. O controle de implementações dos componentes é feito por mapas de interface para implementação, durante o registro dos componentes. Os componentes do VRaptor são registrados primeiro, assim um componente da aplicação sobrescreve a entrada do mapa. Após a fase do registro, os escopos são criados a partir desses mapas.

- **SpringProvider**: o VRaptor procura um `ApplicationContext` do Spring que já esteja registrado, assim consegue aproveitar configurações do Spring que tenham sido registradas num `ContextLoaderListener`, por exemplo. Se não encontrar, o VRaptor procura por um arquivo `applicationContext.xml` no classpath para a configuração inicial. Se também não for encontrado, o VRaptor cria um `ApplicationContext` em

branco. A partir daí o VRaptor usa a API programática do Spring para registrar os componentes, usando os escopos do próprio Spring. Para permitir a sobrescrita, todos os componentes do VRaptor são registrados como infraestrutura e todos os componentes da aplicação são registrados como primários. Assim, se houver mais de um componente implementando uma interface, o da aplicação tem preferência.

- **GuiceProvider**: é criado um `Module` chamado `VRaptorAbstractModule`, que registra todos os componentes do VRaptor e define todos os escopos, que são implementados pelo próprio VRaptor. Então é criado um segundo `Module` que registra todos os componentes da aplicação e sobrescreve o primeiro. Assim, se existirem duas implementações da mesma interface, a da aplicação sobrescreve a do VRaptor.

15.2 Definindo os papéis dos componentes: estereótipos

Após o registro de todos os componentes no provider acontece a definição dos seus papéis: quais serão os controllers, quais serão interceptors, quais serão componentes comuns. Essa definição é feita pela anotação que registra a classe: o estereótipo. Os estereótipos existentes no VRaptor são:

- `@Resource`: define que a classe será um controller e tratará requisições.
- `@Intercepts`: define um `Interceptor`, com possibilidade de configuração da ordem relativa entre interceptors.
- `@Component`: um componente comum do sistema.
- `@Convert`: define um `Converter`, que converte parâmetros de requisição em objetos.
- `@Deserializes`: define um `Deserializer` que, dado um `Content-Type`, deve transformar o corpo da requisição em objetos.

O comportamento de cada um desses estereótipos é definido por componentes que implementam `StereotypeHandler`, que tratam cada uma das classes durante a inicialização do servidor. Se quisermos, é possível criar novos estereótipos na aplicação, com seus respectivos handlers, ou até criar um handler para um estereótipo

padrão do VRaptor. Se quisermos, por exemplo, criar uma lista de todos os controllers do sistema, podemos criar um handler para o estereótipo `@Resource`:

```
@Component
@ApplicationScoped
public class ListaDeControllers implements StereotypeHandler {
  private List<Class<?>> controllers = new ArrayList<>();

  public Class<? extends Annotation> stereotype() {
    return Resource.class;
  }

  public void handle(Class<?> type) {
    controllers.add(type);
  }
}
```

Para criar um estereótipo novo, basta anotar a anotação do estereótipo com `@Stereotype` do VRaptor, e criar o handler respectivo. Todas as anotações com `@Stereotype` registram componentes para injeção de dependências automaticamente.

Como esses `StereotypeHandler`s definem comportamentos importantes da aplicação, eles envolvem outros componentes, que podem ser sobrescritos pela aplicação, se necessário for. Para sabermos o que é possível fazer com cada estereótipo, vamos ver cada um em detalhes.

@Component

Esse estereótipo apenas registra o componente para a injeção de dependências, não possui comportamentos adicionais.

@Convert

O comportamento é dado pelo `ConverterHandler`, que valida se a classe anotada é mesmo um `Converter` e a registra no componente `Converters`, que é o catálogo de todos os converters do sistema.

O componente `Converters` é implementado pela classe `DefaultConverters` e tem por método principal o `to`. Se quisermos uma instância do converter de `Date`, fazemos:

```
Converter<Date> converter = converters.to(Date.class);
```

A ordem de busca dos converters dá preferência para converters registrados pela aplicação, mas caso haja algum problema de prioridade, podemos sobrescrever esse componente para resolver ambiguidades:

```
@Component @ApplicationScoped
public class CustomConverters extends DefaultConverters {
  //delega o construtor

  @Override
  public <T> Converter<T> to(Class<T> clazz) {
    if (clazz.equals(Date.class))
      return container.instanceFor(MeuDateConverterMaroto.class);

    return super.to(clazz);
  }
}
```

@Deserializes

Esse estereótipo recebe a lista de content types que a classe é capaz de deserializar:

```
@Deserializes({"application/json", "text/javascript"})
public class JsonDeserialization implements Deserializer { ... }
```

O seu `StereotypeHandler` é o `DeserializesHandler`, que funciona de forma bastante similar ao `ConverterHandler`: recebe um `Deserializers` no construtor, e monta o catálogo de deserializers, que mapeia content types a classes que os desserializam.

O VRaptor já possui implementações que deserializam JSON e XML, suportando também vendor types, como por exemplo `application/atom+xml` e `application/vnd.myapp+json`. Para formatos diferentes de XML e JSON podemos implementar um deserializer próprio e anotá-lo com `@Deserializes`, passando o content-type suportado.

@Intercepts

O `@Intercepts` é tratado pelo `InterceptorStereotypeHandler`, que registra o `Interceptor` encontrado no `InterceptorRegistry`. Esse componente organiza os interceptors em uma ordem que respeita as restrições que podemos passar ao declarar o `Interceptor`:

```
@Intercepts(before=X.class, after={M.class, N.class})
public class A implements Interceptor { ... }
```

Nesse caso, ele garante uma ordem em que o `A` rode depois do `M` e do `N`, mas antes do `X`. Ordens possíveis seriam `M => N => A => X` e `N => M => A => X`. Os interceptors também são ordenados em relação aos interceptors do VRaptor e, se nada for dito, o interceptor da aplicação executa depois de encontrar o método do controller que será executado na requisição e antes de executar o método do controller.

A implementação padrão desse registro de interceptors é o `TopologicalSortedInterceptorRegistry`, que ordena os interceptors usando a ordem topológica definida pelo `before` e o `after`. Se por acaso houver um ciclo de dependência entre os interceptors, essa implementação identifica na inicialização e gera um erro.

@Resource

É o estereótipo principal, já que define os Controllers da aplicação, e é tratado pelo `ResourceHandler`, que interage com dois componentes: o `RoutesParser`, que extrai rotas a partir de um controller e o `Router`, que guarda todas as rotas do sistema e possibita tanto a geração de uma URL dado um método do controller ou a busca de um método do controller a partir de uma URL e um método HTTP.

A implementação padrão do `RoutesParser` é a `PathAnnotationRoutesParser`, que define as rotas usando as anotações que colocamos no controller, como `@Path`, `@Get` e `@Post`. Essa implementação é a que define a convenção de URL `/nomeDoController/nomeDoMetodo`. Essa e todas as classes do VRaptor que definem convenções foram feitas para serem facilmente sobrescritas.

É bastante difícil implementar um `RoutesParser` do zero, mas é bastante fácil mudar a forma com que a implementação padrão funciona, pois ela possui vários métodos `protected` que são usados durante o algoritmo de criação de rotas. Um exemplo de mudança é se quisermos mudar a convenção para colocar `/admin` antes dos controllers que estão no pacote terminado em `.admin`:

```
@Component @ApplicationScoped
public class AdminRoutesParser extends PathAnnotatedRoutesParser {
  // delega o construtor
```

```
  @Override
  protected String extractControllerNameFrom(Class<?> type) {
    String controllerName = super.extractControllerNameFrom(type);
    if (type.getPackage().getName().endsWith(".admin")) {
      return "admin/" + controllerName;
    }
    return controllerName;
  }

}
```

Como criamos um componente na aplicação que implementa o `RoutesParser`, ela sobrescreverá a implementação do VRaptor. Desse modo, a convenção padrão passa a ser `/admin/nomeDoController/nomeDoMetodo` se o pacote terminar com `.admin`. Podemos sobrescrever outros métodos `protected` dessa classe, como por exemplo:

- `String defaultUriFor(String controllerName, String methodName)`: dado o nome do controller já tratado e o nome do método, retornar a URL padrão.
- `protected boolean isEligible(Method javaMethod)`: decide se vai ser gerada ou não uma rota para o método. Por padrão retorna true para métodos públicos que não são estáticos e nem são os métodos de `Object`.
- `protected String[] getURIsFor(Method javaMethod, Class<?> type)`: dado o método e a classe do controller, retorna a lista de URIs que devem chegar nesse método. Podemos, por exemplo, chamar esse método no `super` e modificar as URIs de alguma forma.
- `protected EnumSet<HttpMethod> getHttpMethods(AnnotatedElement annotated)`: dada uma `Class<?>` ou um `Method`, retornar os métodos HTTP suportados. Esse método só é `protected` a partir do VRaptor 3.5.2.
- `protected List<Route> registerRulesFor(Class<?> baseType)`: dada a classe do controller, retorna todas as rotas possíveis. Um `Route` representa toda a configuração de uma rota, como URI, método http e parâmetros recebidos na URI. Podemos construir uma rota usando o builder disponível a partir do `Router`:

```
Route route = router
  .builderFor("/minha-rota")
  .with(EnumSet.of(HttpMethod.GET))
  .withPriority(Path.DEFAULT)
  .is(classDoController, method)
  .build()
```

Existem muitas possibilidades de mudança, caso tenha alguma dúvida ou idéia que esteja difícil de adaptar usando esses métodos, mande uma mensagem na lista de desenvolvimento do VRaptor: caelum-vraptor-dev@googlegroups.com.

15.3 Tratamento de Requisições

Com todos os estereótipos tratados, o VRaptor termina sua inicialização e o servidor está pronto para tratar requisições. Cada requisição tem como ponto de entrada o filtro do VRaptor (a classe `VRaptor`).

O primeiro componente a ser usado é o `StaticContentHandler`, usado para decidir se a requisição é para um arquivo estático e, então, não tratar a requisição. A implementação padrão verifica apenas se a URI representa um arquivo que está na aplicação, como um javascript ou um css, mas podemos usar esse componente para ignorar outras URLs, já que a declaração de um filtro não possui meios de excluir URLs.

Se for decidido que essa requisição vai ser tratada pelo VRaptor, o provider de injeção de dependências é acionado e, então, inicializa o escopo desta requisição. Para executar de fato a requisição, é usado o componente `RequestExecution`. Esse componente poderia fazer qualquer coisa, dentro do ambiente de injeção de dependências do VRaptor, mas a implementação padrão simplesmente pega todos os `Interceptor`s registrados no sistema e os adiciona no `InterceptorStack`, na ordem definida pelo `before` e `after` de cada `Interceptor`.

Essa `InterceptorStack` controla toda a pilha de execução dos interceptors e é exatamente o objeto que recebemos dentro do método `intercepts` de um `Interceptor` para chamarmos o `stack.next(method, instance)`. Os interceptors só são executados caso o método `accepts` retorne `true`.

A partír daí, todo trabalho é feito por interceptors, que controlam os passos da requisição e algumas das funcionalidades opcionais do VRaptor, como download e upload. Veremos o papel de cada `Interceptor` do VRaptor a seguir.

15.4 Catálogo de Interceptors

Vamos ver a seguir todos os interceptors padrão, numa das ordens possíveis de execução desses interceptors. Como a ordem é topológica, ou seja, apenas respeita as restrições de `before` e `after`, não fixando a ordem de interceptors independentes entre si. Os interceptors da aplicação em geral são inseridos entre o `ResourceLookupInterceptor` e o `ExecuteMethodInterceptor`, mas podemos mudar sua ordem para qualquer lugar, desde que não se crie um ciclo de dependências.

ResourceLookupInterceptor

Esse é o interceptor que procura o método de controller que será executado nessa requisição. Para isso ele usa o componente `UrlToResourceTranslator`, que extrai informações da requisição e procura o `ResourceMethod` no `Router`, usando o método `parse`. Esse método procura por rotas que tratam a URI da requisição e, dentre elas, as que tratam o método HTTP da requisição.

Nesse processo, vários erros podem acontecer:

- Nenhuma rota trata a URI: `ResourceNotFoundException`
- Existe alguma rota que trata a URI, mas não no método HTTP passado: `MethodNotAllowedException` com a lista de métodos HTTP possíveis na URI.
- Existe mais de uma rota trata a mesma URI e método HTTP, na mesma prioridade: `IllegalStateException`, dizendo quais são as rotas conflitantes. Esse erro é da aplicação, e pode ser ajustado com o atributo `priority` da anotação `@Path`:

  ```
  @Path(value = "/{rotaMuitoGenerica}", priority=Path.LOWEST)
  ```

Se tudo der certo, o `ResourceMethod` é guardado dentro do componente `MethodInfo`, para ser usado em todos os outros pontos do sistema, e a requisição continua normalmente.

Em caso de erros, existem dois componentes que os tratam: o `ResourceNotFoundHandler` e o `MethodNotAllowedHandler`, que por padrão retornam os HTTP status 404 e 405, respectivamente, e podem ser sobrescritos por componentes da aplicação, caso não seja esse o comportamento desejado.

ExceptionHandlerInterceptor

É o interceptor que implementa a funcionalidade de tratamento de exceções do VRaptor. Podemos colocar no começo de um método do controller:

```
result.on(MinhaException.class)
     .redirectTo(MeuController.class)
     .metodo();
```

Assim se, durante o resto da requisição a `MinhaException` for lançada, a requisição será redirecionada para o `MeuController.metodo()`.

InstantiateInterceptor

É o interceptor que instancia o controller e o deixa disponível no segundo parâmetro do `stack.next` ou no último parâmetro do método `intercepts`. Esse interceptor não faz nada se a instância do controller já estiver preenchida.

Portanto, se quisermos instanciar o controller de alguma outra forma, devemos criar um interceptor `before=InstantiateInterceptor.class` e passar a instancia criada para o método `stack.next`, ou se quisermos usar a instancia do controller para alguma coisa, devemos criar um interceptor `after=InstantiateInterceptor.class` e usar o `Object` que recebemos no último parâmetro do método `intercepts`.

FlashInterceptor

Implementa o escopo implícito chamado de **Flash**. Esse é o escopo em que colocamos um objeto que será disponibilizado na próxima requisição, após um redirect, por exemplo. Existem duas maneiras de usar esse escopo. A primeira é se incluirmos objetos no `Result` antes de um redirect:

```
result.include("umObjeto", objeto);
result.redirectTo(this).outroMetodo();
```

Todos os objetos incluidos, nesse caso, serão colocados na `HttpSession` e consumidos na próxima requisição, para serem reincluidos no `Result`. Outra maneira de usar esse escopo é passando argumentos para o método redirecionado:

```
result.redirectTo(this).outroMetodo(umParametro, outroParametro);
```

Esses parâmetros são usados para preencher a URI de redirecionamento e serão passados para o `outroMetodo` na próxima requisição. Ou seja, ao invés de usar os parâmetros da requisição para criar os parâmetros do `outroMetodo`, serão usados exatamente os objetos `umParametro` e `outroParametro`. Essa segunda maneira é implementada pelo `LogicResult` e pelo `ParametersInstantiatorInterceptor`.

MultipartInterceptor

Esse interceptor é o responsável pelo upload de arquivos e possui duas implementações: o `CommonsUploadMultipartInterceptor`, baseado na biblioteca **commons-fileupload** e o `Servlet3MultipartInterceptor`, que usa a implementação de upload de servidores compatíveis com Servlet 3. A implementação é selecionada de acordo com o classpath da aplicação.

Se existir o jar do commons-fileupload, a implementação `CommonsUploadMultipartInterceptor` é usada. Senão, o VRaptor verifica a versão de Servlet usada e, caso seja a 3, a implementação `Servlet3MultipartInterceptor` é usada. Em último caso, é usada uma terceira implementação chamada `NullMultipartInterceptor` que nunca será usada (`accepts` sempre retorna `false`) e é impresso um log falando que não existe nenhuma biblioteca de upload disponível.

As duas implementações funcionais do `MultipartInterceptor` recebem o componente `MultipartConfig`, que define duas configurações: o tamanho máximo de uploads, que por padrão é 2MB e uma pasta temporária, onde os uploads poderão ser salvos. Podemos sobrescrever essas configurações criando um `@Component` que implementa `MultipartConfig` na aplicação.

ParametersInstantiatorInterceptor

É o responsável por instanciar os parâmetros dos métodos do controller, de acordo com os parâmetros que vieram na requisição. Para isso ele coordena vários tipos de instanciação de parâmetros. Os passos executados nesse interceptor são:

- Normalização de parâmetros: se mandarmos parâmetros que contém "[]", o VRaptor adiciona índices nesses parâmetros. Dessa forma, se mandarmos:

  ```
  objeto.lista[].atributo=a
  objeto.lista[].atributo=b
  objeto.lista[].atributo=c
  ```

Os parâmetros serão normalizados para:

```
objeto.lista[0].atributo=a
objeto.lista[1].atributo=b
objeto.lista[2].atributo=c
```

- Procura por parâmetros que devem vir do Header HTTP usando a anotação `@HeaderParam`, como em:

```
@Get("/uri")
public void metodo(@HeaderParam("User-Agent") String agent) {
  // ...
}
```

  Esse parâmetro é adicionado como um atributo da requisição.

- Uso do flash: se existirem parâmetros salvos no flash, eles serão usados da maneira que vieram.

- Parsing dos parâmetros da requisição: Se não existirem parâmetros do flash, eles serão instanciados pelo componente `ParametersProvider`. Se ocorrerem erros de conversão nesse passo, eles serão adicionados ao `Validator`, obrigando o método do controller a tratar esses erros.

- Disponibilização dos parâmetros: ao final, os parâmetros são incluídos no `MethodInfo`, para serem usados nos próximos componentes.

O componente `ParametersProvider` é o responsável por pegar todos os parâmetros da requisição e instanciar os parâmetros do método do controller que será executado na requisição. Essa instanciação é feita usando a convenção do VRaptor para preenchê-los:

```
parametro=valor
objeto.atributo=valor
objeto.filho.atributo=valor
lista[0]=valor
lista[0].atributo=valor
```

Essa instanciação é feita chamando o construtor sem argumentos e, então, usando os getters e setters respectivos. Para transformar a String para os objetos de cada atributo são usados os `Converters` que foram registrados. O VRaptor

ainda suporta setar atributos após a conversão, então podemos passar parâmetros parecidos com:

```
pessoa=PessoaFisica
pessoa.cpf=11111111111
pessoa.nome=Joao
```

Se tivermos um converter para `Pessoa`, que instancia uma `PessoaFisica` ou uma `PessoaJuridica` de acordo com o parâmetro que veio, o cpf e o nome serão populados após essa conversão.

Caso queira modificar os parâmetros de alguma maneira, ou simplesmente usá-los, crie um interceptor que roda `after=ParametersInstantiatorInterceptor.class`, receba `MethodInfo` no construtor e use o método `info.getParameters()`.

Existem duas implementações de `ParametersProvider`, uma usando a biblioteca `OGNL` (`OgnlParametersProvider`) e outra usando a biblioteca `IOGI` (`IogiParametersProvider`). A implementação com `OGNL` foi a primeira criada e era a padrão até a versão 3.4 do VRaptor. A implementação que usa o `IOGI` suporta, além do comportamento padrão, o uso de construtor para popular os atributos e injeção de dependências no construtor das classes que são parâmetros de métodos de controller.

Ambas as implementações podem ser personalizadas, sobrescrevendo componentes. No caso do OGNL usando o próprio `OgnlParametersProvider` e seus métodos protected e o componente `OgnlFacade` e no caso do IOGI sobrescrevendo o `VRaptorInstantiator`. Como esse processo de instanciação é bastante complexo, não vou tentar explicá-lo aqui, mas o código dessas classes é razoavelmente legível, principalmente se usarmos os testes de unidades para entender os comportamentos possíveis.

Se for necessário, use a lista de desenvolvimento do VRaptor caelum-vraptor-dev@googlegroups.com para postar dúvidas e ter uma direção sobre qual é a melhor maneira de fazer sobrescritas.

DeserializingInterceptor

É o interceptor que instancia os parâmetros caso o método do controller for anotado com `@Consumes`, passando uma lista de content-types. Para encontrar o `Deserializer` apropriado, usa o componente `Deserializers` e, caso não exista, retorna o status 415 - Unsupported Media Type.

MethodValidatorInterceptor

Valida os parâmetros do método usando Bean Validations, por exemplo:

```
@Post("/uri")
public void metodo(@NotNull String parametro) {...}
```

ExecuteMethodInterceptor

Esse interceptor usa os artefatos produzidos pelos outros interceptors para executar de fato o método do controller. Após a execução, o retorno do método é incluído no `MethodInfo` para ser usado nos próximos interceptors.

Outra característica importante é que, quando fazemos validações nos métodos do controller e chamamos os métodos de redirecionamento do validator, como `validator.onErrorRedirectTo(this).formulario()`, é lançada uma `ValidationException` para evitar que o resto do método seja executado, em caso de erros. Por esse motivo, esse interceptor garante que a `ValidationException` tenha sido lançada em caso de erros e, se não for, lança uma exception dizendo:

```
There are validation errors and you forgot to specify where to go.
Please add in your method something like:"
validator.onErrorUse(page()).of(AnyController.class).anyMethod();
or any view that you like.
If you didn't add any validation error, it is possible that a
conversion error had happened.
```

Na grande maioria dos casos, esse erro acontece por causa de erros de conversão que não foram tratados, portanto sempre verifique os parâmetros passados na requisição caso aconteça esse erro.

DownloadInterceptor

É o interceptor que escreve os dados para download na resposta de métodos do controller que retornam `Download` ou algum objeto que representa dados, como `File`, `InputStream` ou `byte[]`.

OutjectResult

Inclui no `Result` o retorno do método do controller, usando um nome dado pelo componente `TypeNameExtractor`, que transforma nomes de classes em

nome de objetos. A convenção do VRaptor para isso é o nome da classe com a primeira minúscula, para tipos simples, e o nome da classe seguido de `List`.

```
@Get("/uri")
public Livro metodo() { ... }
// equivalente a result.include("livro", retornoDoMetodo);

@Get("/uri")
public List<Livro> metodo() { ... }
// equivalente a result.include("livroList", retornoDoMetodo);
```

Essa convenção pode ser mudada sobrescrevendo o `TypeNameExtractor`. Cuidado, pois esse componente é usado também em outros lugares.

ForwardToDefaultViewInterceptor

É o último interceptor da pilha. Seu objetivo é redirecionar para a jsp padrão caso o `Result` não tenha sido usado para alterar o resultado. Esse interceptor não chama `stack.next`, então não adianta criar interceptors que rodem depois dele.

Para mudar a convenção do caminho da JSP padrão, podemos sobrescrever o componente `PathResolver`. A sua implementação padrão também foi feita para ser sobrescrita facilmente, sem ter que implementar todo o algoritmo. Por exemplo, se quisermos trocar de jsp para velocity, podemos criar o componente:

```
@Component
public class VelocityPathResolver extends DefaultPathResolver {
  //delega o construtor
  @Override
  protected String getExtension() {
    return "vm";
  }
  @Override
  protected String getPrefix() {
    return "/WEB-INF/vms/"
  }
}
```

15.5 Mais informações sobre o funcionamento do VRaptor

O VRaptor possui uma suíte de testes bastante abrangente e legível, feita para garantir o funcionamento esperado dos componentes, mas também para documentar cada uma das funcionalidades implementadas.

Outra maneira de entender melhor o funcionamento é tentando implementar alguma funcionalidade para o VRaptor. **Pull Requests** e sugestões de funcionalidades são muito bem vindas e são ótimas maneiras de aprender a desenvolver códigos bastante diferentes do que a gente faz no dia a dia de aplicações. Um código de framework/biblioteca tem desafios únicos e bastante interessantes e isso aumenta o seu conhecimento.

Contribua para o VRaptor ou para qualquer outra biblioteca open source que você usa nas suas aplicações, e se torne um desenvolvedor melhor, além de ser reconhecido por desenvolvedores no mundo inteiro.